

Guidewire ClaimCenter™

Integration Guide

Release: 10.2.4



© 2024 Guidewire Software, Inc.

For information about Guidewire trademarks, visit <https://www.guidewire.com/legal-notices>.

Guidewire Proprietary & Confidential — DO NOT DISTRIBUTE

Product Name: Guidewire ClaimCenter

Product Release: 10.2.4

Document Name: Integration Guide

Document Revision: 10-December-2024

Contents

Support.....	23
---------------------	-----------

Part 1

Planning integration projects.....	25
---	-----------

1 Overview of integration components.....	31
Preparing for integration development.....	31
Regenerating integration libraries and WSDL.....	33
Regenerating the Java API libraries.....	33
WS-I web service regeneration of WSDL for local GUnit tests.....	33
Required files for integration programmers.....	33
Public IDs and integration code.....	34
Creating your own public IDs.....	35
Guidewire internal classes and methods.....	35
2 Proxy servers.....	37
Configuring a proxy server with Apache HTTP Server.....	38
Install Apache HTTP Server: Basic checklist.....	38
Certificates, private keys, and passphrase scripts.....	39
Proxy server integration types.....	39
ISO proxy communication.....	39
Metropolitan proxy communication.....	39
Bing maps geocoding service communication.....	39
Proxy building blocks.....	40
Downstream proxy with no encryption.....	40
Downstream proxy with encryption.....	41
Upstream (reverse) proxy with encryption for service connections.....	41
Upstream (reverse) proxy with encryption for user connections.....	42
Modify the server to receive incoming SSL requests.....	43
3 Java and OSGi support.....	45
Implementing plugin interfaces in Java and optionally OSGi.....	45
Choosing between a Java plugin and an OSGi plugin.....	46
Your IDE options for plugin development in Java.....	46
Java IDE inspections that flag unsupported internal APIs.....	47
Install the internal API inspection in IntelliJ IDEA.....	47
Accessing entity data from Java.....	48
Regenerate Java API libraries.....	49
Java API reference documentation.....	49
Accessing entity instances from Java.....	49
Accessing typecodes from Java.....	50
Getting a reference to an existing bundle from Java.....	50
Creating a new bundle from Java.....	50
Creating a new entity instance from Java.....	51
Querying for entity data from Java.....	51
Using reflection to access Gosu classes from Java.....	51
Using Gosu enhancement properties and methods from Java.....	53

Class loading and delegation for non-OSGi Java.....	53
Deploying non-OSGi Java classes and JAR files.....	54
Deploy an example Java class with no plugins and no entities.....	55
Using IntelliJ IDEA with OSGi editor to deploy an OSGi plugin.....	56
Set up a project with an OSGi plugin module.....	56
Create an OSGi-compliant class that implements a plugin interface.....	57
Example startable plugin in Java using OSGi.....	58
About the OSGi Ant build script.....	59
Compile and install your OSGi plugin as an OSGi bundle.....	60
Configuring third-party libraries in an OSGi plugin.....	61
Advanced OSGi dependency and settings configuration.....	65
OSGi dependencies.....	65
OSGi build properties.....	65
OSGi bundle metadata configuration file.....	65
OSGi build configuration Ant script.....	66
Update your OSGi plugin project after product location changes.....	66
Part 2	
Web services.....	67
4 Publishing web services.....	71
Web service publishing quick reference.....	71
Web service publishing annotation reference.....	73
Data transfer objects.....	75
Publishing and configuring a web service.....	76
Declaring the namespace for a web service.....	76
Specifying the minimum run level for a web service.....	77
Specifying required permissions for a web service.....	77
Overriding a web service method name or visibility.....	77
Web service invocation context.....	78
Web service class life cycle and variables scoped locally to a request.....	79
Generating and publishing WSDL.....	80
Example WSDL.....	81
Calling a ClaimCenter web service from Java.....	82
Generate Java classes that make SOAP client calls to a SOAP API.....	83
Using the generated Java classes.....	84
Adding HTTP basic authentication in Java.....	84
Adding SOAP header authentication in Java.....	85
Testing web services with local WSDL.....	86
Using your class to test local web services.....	86
Writing unit tests for your web service.....	86
Web services authentication plugin.....	87
Writing an implementation of the web services authentication plugin.....	88
Login authentication confirmation.....	88
Request or response XML structural transformations.....	89
Transforming a generated schema.....	89
Converting a Gosu object to and from XML.....	89
Using predefined XSD and WSDL.....	92
Adding an invocation handler for preexisting WSDL.....	92
Example of an invocation handler for preexisting WSDL.....	93
Invocation handler responsibilities.....	94

Referencing additional schemas in your published WSDL.....	95
Validating requests using additional schemas as parse options.....	96
Creating an XML Schema JAR file.....	96
Setting response serialization options, including encodings.....	99
Adding advanced security layers to a web service.....	99
Transformations on data streams.....	100
Using WSS4J for encryption, signatures, and other security headers.....	101
Locale and language support.....	104
Checking for duplicate external transaction IDs.....	104
Exposing typelists and enums as enumeration values and string values.....	105
Optional stateful session affinity using cookies.....	105
5 Calling web services from Gosu.....	107
Loading WSDL locally by using web service collections.....	108
Loading WSDL directly into the file system for testing purposes.....	108
Types of client connections.....	109
How Gosu processes WSDL.....	109
Web service API objects are not thread safe.....	110
Working with web service data by using Gosu XML APIs.....	111
What Gosu creates from your WSDL.....	111
Special behavior for multiple ports.....	111
Request XML complexity affects appearance of method arguments.....	112
Adding configuration options to a web service.....	112
Directly modifying the WSDL configuration object for a service.....	113
Authentication schemes for consuming WS-I web services from Gosu.....	114
Guidewire suite configuration file.....	116
Setting Guidewire transaction IDs on web service requests.....	117
Setting a timeout on a web service.....	118
Custom SOAP headers.....	118
Server override URL.....	118
Setting XML serialization options.....	119
Setting locale or language in a Guidewire application.....	119
Implementing advanced web service security with WSS4J.....	119
One-way methods.....	121
Asynchronous method calls to web services.....	121
MTOM attachments with Gosu as web service client.....	122
6 General web services.....	123
Importing administrative data.....	123
Prepare exported administrative data for import.....	123
Importing prepared administrative data.....	123
CSV import and conversion.....	124
Advanced import or export.....	124
Maintenance tools web service.....	124
Using web service methods with batch processes.....	124
Using web service methods with work queues.....	126
Using web service methods with startable plugins.....	127
Using maintenance tools web services to archive or restore claims.....	128
Marking claims for purging by using web services.....	128
Changing a Contact subtype by using web services.....	128
Recalculating aggregate limits by using web services.....	129
Getting the calculation date of team statistics by using web services.....	129
Mapping typecodes and external system codes.....	129

The TypelistToolsAPI web service.....	130
The TypecodeMapper class.....	131
Data destruction web service.....	132
PersonalDataDestructionAPI web service methods.....	133
Lifecycle of a personal data destruction request.....	133
Personal data destruction request entities.....	134
Example of a request made with AddressBookUID.....	135
Example of a request made with PublicID.....	135
Typelists for status of personal destruction workflow.....	135
Work queues used in personal data destruction.....	136
PersonalDataDestructionController class.....	137
The Profiler API web service.....	137
Server state web service.....	139
System tools web service.....	140
Getting and setting the run level.....	140
Getting server and schema versions.....	141
Clustering tools.....	141
Table import web service.....	142
Template web service.....	144
Workflow web service.....	145
Zone data import web service.....	146
7 Claim-related web services.....	149
Claim web service APIs and data transfer objects.....	149
Date- and time-related DTOs.....	149
Adding first notice of loss from external systems.....	150
Getting a claim from external systems.....	151
Importing a claim from XML from external systems.....	151
Migrating a claim from external systems.....	151
Getting information from claims/exposures from external systems.....	151
Claim bulk validate from external systems.....	152
Bulk load and validate claims.....	152
Previewing assignments from external systems.....	153
Closing and reopening a claim from external systems.....	153
Policy refresh from external systems.....	153
Add user permissions on a claim from external systems.....	153
Archiving claims from external systems.....	154
Restoring archived claims from external systems.....	154
Managing activities from external systems.....	155
Adding a contact from external systems.....	156
Adding a document from external systems.....	156
Adding an exposure from external systems.....	156
Getting an exposure from external systems.....	157
Closing and reopening an exposure from external systems.....	157
Adding to claim history from external systems.....	157
Creating a note from external systems.....	157
8 Service request integration.....	159
Service request web service (ServiceRequestAPI) overview.....	159
Search for service requests.....	159
Get service request.....	160
Update service request reference number.....	160
Get document content for a service request.....	160

Service request messages.....	161
Add or replace quote on service request.....	161
Adding invoices.....	162
Update expected quote completion date for a service request.....	163
Update expected service completion date.....	163
Add document to a service request.....	163
Accept work for a service request.....	164
Decline work for a service request.....	164
Record work resumed for service request.....	164
Record waiting for service request.....	164
Record work completed for service request.....	165
Cancel service request.....	165
Withdraw invoice for service request.....	165
Statement creation instructions reference.....	166
Part 3	
Plugins.....	167
9 Overview of plugins.....	169
Implementing plugin interfaces.....	169
Choose a plugin implementation type.....	169
Writing a plugin implementation class.....	170
Registering a plugin implementation class.....	172
Deploying Java files, including Java code called from Gosu plugins.....	173
Error handling in plugins.....	173
Enabling or disabling a plugin.....	174
Example Gosu plugin.....	174
Special notes for Java plugins.....	175
Getting plugin parameters from the plugins registry editor.....	175
Getting the local file path of the root and temp directories.....	175
Plugin registry APIs.....	175
Plugin thread safety.....	176
Using Java concurrent data types, even from Gosu.....	177
Avoiding singletons because of thread-safety issues.....	178
Design plugin implementations to support server clusters.....	179
Reading system properties in plugins.....	179
Do not call local web services from plugins.....	180
Creating unique numbers in a sequence.....	180
Restarting and testing tips for plugin developers.....	180
Summary of ClaimCenter plugins.....	180
10 Authentication integration.....	187
Overview of user authentication interfaces.....	187
User authentication source creator plugin.....	188
Handling errors in the authentication source plugin.....	189
Create a custom exception type.....	189
Authentication data in HTTP attributes.....	189
Authentication data in parameters in the URL.....	190
Authentication data in HTTP headers for HTTP basic authentication.....	190
User authentication service plugin.....	191
Authentication service sample code.....	191
Handling errors in authentication service plugins.....	192

SOAP API user permissions and special-casing users.....	193
Example authentication service authentication.....	193
Deploying user authentication plugins.....	193
Database authentication plugins.....	194
Configuration for database authentication plugins.....	196
WS-I web services authentication.....	196
11 Document management.....	197
Document storage overview.....	197
Storing document content and metadata.....	198
Document storage plugin architecture.....	198
Understanding document IDs.....	199
Document IDs in emails and notes.....	200
Implementing a document content source for external DMS.....	200
Check for document existence.....	201
Add documents and metadata.....	201
Retrieve documents.....	203
Update documents and metadata.....	205
Remove documents.....	205
User interface elements when document management system unavailable.....	205
Render a document.....	206
Implementing an IDocumentMetadataSource plugin.....	206
Basic methods for the IDocumentMetadataSource plugin.....	207
Linking methods for the IDocumentMetadataSource plugin.....	209
Document storage plugins in the base configuration.....	210
Documents storage directory and file name patterns.....	210
Remember to store public IDs in the external system.....	211
Asynchronous document storage.....	212
Configure asynchronous document storage (sync-first or async-only).....	215
Disable asynchronous document content storage completely (sync-only).....	216
Errors and document validation in asynchronous document storage.....	216
Final documents.....	217
Final documents with IDocumentMetadataSource enabled.....	217
Final documents with IDocumentMetadataSource disabled.....	217
Document interactions also depend on user permissions.....	218
APIs to attach documents to business objects.....	218
Gosu APIs to attach documents to business objects.....	218
Web service APIs to attach documents to business objects.....	218
Document management servlet for testing vendor documents.....	219
DMS servlet plugins and ContactManager.....	219
Document DTO used by the DMS servlet.....	221
Configuring the DMS servlet in ContactManager.....	221
REST APIs of the DMS servlet.....	221
Related APIs for DMS servlet.....	223
12 Document production.....	225
Document production types.....	226
Understanding synchronous and asynchronous document production.....	226
User interface flow for document production.....	226
Document production plugins.....	229
Implementing synchronous document production.....	230
Implementing asynchronous document production.....	231
Configuring document production implementation mapping.....	231

Add a custom MIME type for document production.....	232
Licensing for server-side document production.....	232
Licensing for Microsoft Excel and Word document production.....	232
Licensing for PDF document production.....	233
Write a document template descriptor and install a template.....	233
Example simple template descriptor XML file with no context objects.....	234
Example template descriptor XML file with context objects.....	235
Form fields and field groups.....	237
Field groups.....	238
Display values.....	238
Context objects.....	240
Gosu APIs on template descriptor instances.....	242
Template descriptor fields related to form fields and values to merge.....	243
XML format for document template descriptors.....	243
Add custom attributes document template descriptor XML format.....	244
Create your actual template file.....	245
Adding document templates and template descriptors to a configuration.....	246
Creating new documents from Gosu rules.....	247
Example document creation after sending an email.....	247
Important notes about cached document UIDs.....	248
13 Geographic data integration.....	249
Geocoding plugin integration.....	249
How ClaimCenter uses geocode data.....	249
What the geocoding plugin does.....	250
Synchronous and asynchronous calls to the geocoding plugin.....	250
Using a proxy server with the geocoding plugin.....	250
Batch geocoding only some addresses.....	250
Base implementation of the Bing maps geocoding plugin.....	251
Geocoding and routing responses.....	251
Geocoding request generators.....	251
Deploy a geocoding plugin.....	252
Writing a geocoding plugin.....	253
Using the abstract geocode Java class.....	253
High-level steps to writing a geocoding plugin implementation.....	254
Geocoding an address.....	254
Getting driving directions.....	256
Getting a map for an address.....	257
Getting an address from coordinates (reverse geocoding).....	258
Geocoding status codes.....	258
List of geocoding status codes.....	258
14 Reinsurance integration.....	261
Reinsurance plugin.....	261
Enabling the reinsurance plugin for production.....	261
15 Encryption integration.....	263
Using encrypted properties.....	263
Setting encrypted properties.....	263
Querying encrypted properties.....	264
Sending encrypted properties to other systems.....	265
Setting up encryption.....	266
Writing your encryption plugin.....	267
Example encryption plugin.....	267

Changing your encryption algorithm.....	269
Change your encryption algorithm.....	270
Installing your encryption plugin.....	271
Encryption features for staging tables.....	271
Encrypt any encrypted properties in staging tables.....	272
16 Free-text search integration.....	273
Overview of free-text search plugins.....	273
Connecting the free-text plugins to the Guidewire Solr Extension.....	273
Enabling and disabling the free-text plugins.....	274
Running the free-text plugins in debug mode.....	274
Free-text load and index plugin and message transport.....	274
Message destination for free-text search.....	275
Enable the message destination for free-text search.....	275
ISolrMessageTransportPlugin plugin parameters.....	275
Free-text search plugin.....	276
ISolrSearchPlugin plugin parameters.....	276
17 Management integration.....	277
Management plugin examples.....	278
Calling the example JMXManagementPlugin.....	279
Prepare the JMXManagementPlugin implementation.....	279
Register the JMXManagementPlugin implementation.....	280
Calling the JMXManagementPlugin from an external application.....	280
18 Other plugin interfaces.....	283
SharedBundlePlugin marker interface.....	283
Credentials plugin.....	283
Preupdate handler plugin.....	284
Validation plugin.....	286
Work item priority plugin.....	287
Exception and escalation plugins.....	287
Sending emails.....	287
IEmailTemplateSource plugin.....	290
Defining base URLs for fully qualified domain names.....	291
Claim number generator plugin.....	292
Cancelling claim numbers.....	292
Approval plugin.....	293
Automatic address completion and fill-in plugin.....	293
Phone number normalizer plugin.....	293
Official IDs mapped to tax IDs plugin.....	295
Testing clock plugin (for non-production servers only). Using the testing clock.....	295 296
19 Startable plugins.....	299
Starting a startable plugin.....	299
Initializing a startable plugin.....	300
Registering startable plugins.....	300
Manually starting and stopping a startable plugin.....	300
Writing a singleton startable plugin.....	301
User contexts for startable plugins.....	302
Simple startable plugin example.....	302
Writing a distributed startable plugin.....	303
Defining startable plugins in Java.....	305
Persistence and startable plugins.....	305

20 Multi-threaded inbound integration.....	307
Inbound integration core plugin interfaces.....	307
Inbound integration handlers for file and JMS integrations.....	308
Configuring inbound integration.....	309
Thread pool configuration XML elements.....	310
Varying the inbound integration settings.....	311
Inbound integration configuration XML elements.....	311
Using the inbound integration polling and throttle intervals.....	315
Inbound file integration.....	316
Create an inbound file integration.....	317
Example of an inbound file integration.....	318
Inbound JMS integration.....	319
Create an inbound JMS integration.....	320
Example of an inbound JMS integration.....	321
Custom inbound integrations.....	323
Writing a custom inbound integration plugin.....	323
Writing a work agent implementation.....	324
Install a custom inbound integration.....	329
21 Redis integration.....	331
Using Redis with a Guidewire cluster.....	331
Set up Guidewire cluster with Redis server.....	332
Example Redis setup.....	333
Redis plugin parameters.....	333
Accessing Redis information.....	335

Part 4

Messaging..... 337

22 Messaging and events.....	339
Overview of messaging flow.....	341
Messaging flow details.....	342
Overview of message destinations.....	346
Use the Messaging editor to create new messaging destinations.....	347
Sharing plugin classes across multiple destinations.....	349
Handling acknowledgments.....	349
Destinations in the base configuration.....	350
Message processing.....	351
Message processing cycle.....	351
Messaging database transactions during sending.....	353
Messaging plugin interaction and flow diagram.....	361
Messaging events in ClaimCenter.....	362
List of messaging events.....	363
Triggering a remove-related event.....	370
Triggering custom events.....	371
Custom events from SOAP acknowledgments.....	371
How custom events affect pre-update and validation.....	371
Events for contact changes.....	371
Events for special subobjects.....	373
Ordering events.....	373
No events from import tools.....	374
Generating new messages in Event Fired rules.....	374

Rule set structure.....	374
Simple message payload.....	375
Multiple messages for one event.....	375
Determining what changed.....	375
Rule sets must never call message methods for ACK, error, or skip.....	376
Save values across rule set executions.....	376
Creating a payload by using Gosu templates.....	377
Setting a message root object or primary object.....	377
Creating XML payloads by using GX models.....	379
Using Java code to generate messages.....	379
Saving attributes of a message.....	379
Handling policy changes on a claim.....	380
Maximum message size.....	380
If multiple events fire, which message sends first?.....	380
Restrictions on entity data in messaging rules and messaging plugins.....	380
Event fired rule set restrictions for entity data changes.....	380
Messaging plugin restrictions for entity data changes.....	381
Database transactions when creating messages.....	382
Messaging plugins must not call SOAP APIs on the same server.....	382
Delaying or censoring message generation.....	382
Detect claim state changes, such as from draft to open.....	382
Validity and rule-based event filtering.....	383
Late binding data in your payload.....	383
Implement late binding.....	384
Tracking a specific entity with a message.....	385
Implementing message plugins.....	385
Ensuring thread safety in messaging plugin code.....	385
Initializing a messaging plugin.....	386
Getting messaging plugin parameters from the plugin registry.....	386
Implementing message payload transformations before send.....	386
Implementing a message transport plugin.....	388
Implementing post-send processing.....	389
Implementing a MessageReply plugin.....	390
Error handling in messaging plugins.....	392
Suspend, resume, and shut down a messaging destination.....	393
Map message payloads by using tokens.....	394
Message status code reference.....	395
Reporting acknowledgments and errors.....	397
Message sending error behaviors.....	397
Submitting ACKs, errors, and duplicates from messaging plugins.....	398
Using web services to submit ACKs and errors from external systems.....	399
Using web services to retry messages from external systems.....	399
Message error handling.....	400
Resynchronizing messages for a primary object.....	401
Cloning new messages from pending messages.....	403
Resync and ClaimCenter financials.....	403
How resynchronization affects preupdate and validation.....	403
Resync in ContactManager.....	403
Monitoring messages.....	404
Messaging tools web service.....	404
Acknowledging messages.....	404

Getting the ID of a message.....	405
Retrying messages.....	405
Skiping a message.....	406
Resynchronizing a claim for a destination.....	407
Purging completed messages.....	407
Suspending a destination.....	407
Resuming a destination.....	408
Getting messaging statistics.....	409
Obtaining the status of a destination.....	409
Getting configuration information from a destination.....	410
Changing messaging destination configuration parameters.....	411
Included messaging transports.....	412
Email message transport.....	412
Register and enable the console message transport.....	412

Part 5

Financials.....**415**

23 Financials integration.....	417
Financial transaction status and status transitions.....	417
Status transitions for financial transactions.....	418
Check and transaction status transitions and message acknowledgment	419
In event fired rules, treat status transition as final.....	421
Debugging financials messaging.....	421
Claim financials web service data transfer objects.....	421
Claim financials web service (ClaimFinancialsAPI).....	421
Importing processed financials from external systems.....	422
Creating new financials from external systems.....	422
Multicurrency with new financials.....	423
Multicurrency foreign exchange adjustment SOAP APIs.....	424
Check integration.....	424
Required message acknowledgments for checks.....	429
Updating check status.....	429
Voiding, stopping, denying, and reissuing checks.....	430
Modifying the check scheduled send date.....	432
Instant check integration.....	433
Payment transaction integration.....	436
Required message acknowledgments for payments.....	439
Integration events for payment recoding.....	439
Integration payment events for check transfer.....	441
Payment integration properties for event fired rules.....	441
Recovery reserve transaction integration.....	442
Required message acknowledgments for recovery reserves.....	443
Recovery transaction integration.....	443
Required message acknowledgments for recoveries.....	444
Denying recoveries.....	444
Reserve transaction integration.....	444
Required message acknowledgments for reserves.....	445
Bulk invoice integration.....	445
Bulk invoice validation.....	446
Bulk invoice web service APIs.....	447

Post-submission actions on a bulk invoice.....	451
Bulk invoice processing performance.....	452
Bulk invoice (top level entity) status transitions.....	453
Required message acknowledgments for bulk invoices.....	459
Bulk invoice status changes using SOAP or domain method.....	459
Bulk invoice item status transitions.....	459
Bulk invoice batch processes.....	462
Deduction plugins.....	462
Deduction calculations for checks.....	462
Handling other deductions.....	463
Part 6	
Claim and policy integrations.....	465
24 Claim and policy integration.....	467
Policy system notifications.....	467
General purpose notification system.....	467
Large loss notification implementation details.....	468
Enabling large loss notification.....	469
Using a policy administration system other than PolicyCenter.....	469
Add other notification types.....	470
Policy search plugin.....	472
Typical chronological flow of policy search and retrieval.....	473
Policy search example and additional information.....	474
Policy retrieval additional information.....	475
Modifying the New Claim wizard to reload a policy.....	477
Claim search web service for policy system integration.....	478
Search for claims.....	478
Get number of matched claims.....	479
Get claim detail.....	479
User claim view permission.....	479
ClaimCenter exit points to PolicyCenter and ContactManager.....	479
PolicyCenter Product Model import into ClaimCenter.....	480
Configuring the ClaimCenter Typelist Generator.....	482
Run the ClaimCenter Typelist Generator.....	486
Using generated typelists in ClaimCenter.....	487
Typelist localization.....	492
Catastrophe policy location download.....	493
Catastrophe policy location overview.....	493
Catastrophe areas of interest data model.....	493
Catastrophe policy location download processing.....	495
Policy location search plugin.....	496
Policy refresh overview.....	497
Preserving relationships among claims and policies with policy refresh.....	497
Matching overview.....	498
Differences between policy refresh and policy select.....	499
How policy refresh handles special situations.....	499
Policy refresh plugins and configuration classes.....	500
Policy refresh configuration base class.....	501
Policy refresh steps and associated implementation.....	502
Determining the extent of the policy graph.....	504

The policy graph within a claim.....	504
The policy refresh plugin determines the objects in the policy graph.....	504
Extending the data model for the policy graph.....	505
Policy refresh entity matcher details.....	505
Matching related objects.....	505
Registering matcher classes.....	506
Best practices for entity matcher classes.....	506
Entity matcher classes in the base configuration.....	507
Example of changing matching criteria.....	508
Policy refresh relinking details.....	509
Example relink handler class for contact tag objects.....	510
Relink filters.....	510
Policy refresh policy comparison display.....	513
Difference objects.....	513
Difference display objects.....	514
Configuring visibility in the comparison tree.....	519
Construction of the policy comparison tree.....	520
Configuring labels and display order.....	521
Changing icons for added, removed, unchanged, and changed.....	523
Policy refresh configuration examples.....	523
Adding a claim-specific entity to policy refresh.....	523
Adding a policy-specific entity to policy refresh.....	524
Modifying your policy system for changed entities in the policy graph.....	525
Configuring policy refresh for new policy-specific entities.....	527
Part 7	
Contact integrations.....	531
25 Contact integration.....	533
Integrating with a contact management system.....	533
Inbound contact integrations.....	534
Asynchronous messaging with the contact management system.....	534
ClaimCenter synchronizing and mapping.....	535
Retrieve a contact.....	535
Retrieve a persistable contact.....	536
Contact searching.....	536
Finding duplicate contacts.....	537
Find duplicate contacts.....	538
Adding contacts to the external system.....	538
Updating contacts in the external system.....	538
Configuring contact links.....	539
Contact web service APIs.....	540
Deleting a contact.....	540
Updating a contact.....	540
Merging contacts.....	540
Handling rejection and approval of pending changes.....	541
Part 8	
Importing claims data.....	543

26 Importing from database staging tables.....	545
Files for zone data import.....	547
System parameters for database import.....	548
Database import tables and columns.....	548
Staging tables.....	548
Load error table.....	549
Exclusion table.....	549
Load history table.....	549
Load command IDs.....	549
Load user IDs.....	550
Load time stamps.....	550
Import tools and commands.....	550
Server modes and run levels for database staging table import.....	551
Database consistency checks.....	551
Your conversion tool.....	552
Integrity checks.....	552
Database performance considerations.....	553
Loading zone data into staging tables.....	554
Clearing errors from staging tables.....	554
Importing data into operational tables.....	555
Using a conversion tool to populate the staging tables.....	556
Important properties and concepts for database import.....	557
Handling errors in database import using a conversion tool.....	559
Loading ClaimCenter entities.....	559
Detailed work flow for a typical database staging table import.....	562
Prepare the data and the ClaimCenter database.....	562
Import data into the staging tables.....	563
Handling encrypted properties.....	564
Load staging table data into operational tables.....	565
Perform post-import tasks.....	566
Table import tips and troubleshooting.....	566
27 FNOL mapper.....	567
Importing ACORD XML data with the FNOL mapper.....	567
Custom mapping of ACORD XML data.....	567
Special handling of ACORD XML data.....	567
Importing custom XML data with the FNOL mapper.....	568
FNOL mapper detailed flow.....	568
Structure of FNOL mapper classes.....	568
Using Gosu native XML support with FNOL mappers.....	569
Instantiating address, contact, exposure, or incident mapping classes.....	569
Configuration files and typecode mapping for ACORD data.....	570
Additional classes and enhancements for FNOL mapping.....	571
FNOL mapper and built-in ACORD class diagram.....	572
Example FNOL mapper customizations.....	574
Add mapping for a new exposure type.....	574
Add additional properties to an exposure.....	575
Create a new mapper for non-ACORD data.....	575

Part 9**ISO and Metropolitan.....** **577**

28 Insurance services office (ISO) integration.....	579
ISO integration overview.....	579
Claim-based messaging with legacy support for exposures.....	579
Match reports.....	581
Implementation overview.....	581
Reference material for ISO integration work.....	582
What to know about ISO mapping.....	582
Implementation technical overview of ClaimSearch web service.....	583
Enable ISO integration.....	584
ISO implementation checklist.....	585
ISO network architecture.....	587
Basic ISO message types.....	587
ISO network layout and URLs.....	588
ISO and ClaimCenter clusters.....	590
ISO activity and decision timeline.....	591
Initial ISO validation.....	591
Simple claim change.....	591
Simple key field change.....	592
ISO activity and decision diagrams.....	592
ISO authentication and security.....	596
ISO security with customer IDs and IP ranges.....	596
ISO security with customer passwords.....	596
ISO proxy server setup.....	598
ISO validation level.....	598
ISO messaging destination.....	599
ISO receive servlet and the ISO reply plugin.....	601
ISO properties on entities.....	601
ISO user interface.....	602
ISO properties file.....	603
Populating match reports from ISO's response.....	605
Converting from ClaimContact (ClaimCenter) to ClaimParty (ISO).....	606
ISO type code and coverage mapping.....	607
Coverage mapping.....	607
ISO payload XML customization.....	608
ISO payload generation properties reference.....	609
ISO match reports.....	610
ISO exposure type changes.....	611
ISO date search range and resubmitting exposures.....	612
ISO integration troubleshooting.....	612
ISO formats and feeds.....	613
Update the XML files provided by ISO.....	614
29 Metropolitan Reporting Bureau integration.....	617
Overview of ClaimCenter integration with Metropolitan Reporting Bureau.....	617
ClaimCenter Metropolitan integration architecture.....	619
Metropolitan configuration.....	620
Enable Metropolitan configuration.....	620
Metropolitan properties file.....	620
ClaimCenter display keys for Metropolitan reports.....	621
Configure activity patterns.....	621
Configure the messaging plugin retries.....	622
Metropolitan report templates and report types.....	622

Metro report types and loss types.....	622
Edit Gosu template files.....	623
Map report types to Gosu template files.....	623
Metro report template special cases.....	623
Add new report types.....	624
Metropolitan entities, typelists, properties, and statuses.....	624
MetroReport entity.....	625
MetroReportType typelist.....	625
MetroAgencyType typelist.....	625
Document entity.....	625
Metropolitan report statuses and workflow.....	625
Customizing Metropolitan timeouts.....	627
Metropolitan error handling.....	628
Part 10	
Data transfer structures.....	629
30 Guidewire InsurancePlatform Integration Views.....	631
Overview of Integration Views.....	631
Creating an Integration View.....	632
Mapping a data object to an Integration View schema.....	632
Creating a schema for Integration View objects.....	633
Filtering Integration View objects for select attributes.....	635
Accessing and naming files related to Integration Views in Guidewire Studio.....	635
Handling an Integration View.....	635
Handling an Integration View using a REST API.....	635
Handling an Integration View using an event message handler.....	636
31 GX models.....	639
Create a GX model.....	639
Including a GX model inside another GX model.....	641
Mappable and unmappable properties.....	642
Normal and key properties.....	642
Automatic publishing of the generated XSD schema.....	642
Create an instance of a GX model.....	643
GXOptions.....	643
GX model labels.....	644
Assign a label to a GX model property.....	645
Reference a GX model label.....	645
Create an instance of a GX model with labels.....	645
GX model label example.....	647
Serialize a GX model object to XML.....	647
Arrays of entities in XML output.....	647
Sending a message only if data model fields changed.....	648
Conversions from Gosu types to XSD types.....	649
Converting GX models to Integration Views.....	650
32 Archiving integration.....	653
Overview of archiving integration.....	653
Basic integration flow for archiving storage.....	653
Basic integration flow for archiving retrieval.....	654
Upgrading the data model of retrieved data.....	654
Contact info entities and other archiving entities that persist.....	655

Error handling during archive.....	655
Archiving storage integration detailed flow.....	655
Archiving retrieval integration detailed flow.....	657
Archive source plugin.....	658
Archive source plugin methods and archive transactions.....	659
Archive source plugin storage methods.....	659
Archive source plugin retrieval methods.....	661
Archive source plugin utility methods.....	662
33 Servlets.....	667
Implementing servlets.....	667
@Servlet annotation.....	668
Servlet definition in servlets.xml.....	669
Important HttpServletRequest object properties.....	670
Create and test a basic servlet.....	671
Example of a basic servlet.....	672
Implementing servlet authentication.....	672
Create a servlet that provides basic authentication.....	673
Test your basic authentication servlet.....	673
Example of a basic authentication servlet.....	674
Supporting multiple authentication types.....	675
Abstract HTTP basic authentication servlet class.....	675
Abstract Guidewire authentication servlet class.....	675
ServletUtils authentication methods.....	676
Example of a servlet using multiple authentication types.....	677
Test your servlet using multiple authentication types.....	678
34 Data extraction integration.....	679
Data extraction using web services.....	680
Using Gosu templates for data extraction.....	681
Gosu template APIs for data extraction integration.....	682
35 Inbound files integration.....	683
Work queues.....	683
InboundFilePurgeWorkQueue.....	684
InboundChunkWorkQueue.....	684
Implementing an integration.....	684
Create an InboundFileHandler implementation.....	684
Configuring inbound files integration.....	686
36 Outbound files integration.....	691
Work queues.....	691
OutboundFilePurgeWorkQueue.....	692
OutboundRecordPurgeWorkQueue.....	692
Implementing an integration.....	692
Create an OutboundFileHandler implementation.....	692
Configure outbound files integration.....	693
Part 11	
Custom processing by using work queues and batch processes.....	697
37 Custom processing.....	699
Overview of custom processes.....	699
Processing modes.....	699
Choosing a mode for a custom process.....	700

About scheduling ClaimCenter processes.....	700
About manually executing ClaimCenter processes.....	701
Batch processing typecodes.....	701
Custom work queues.....	701
About custom work queue classes.....	702
Work queues and work item entity types.....	702
Work queues that use StandardWorkItem.....	703
Lifecycle of a work item.....	703
Considerations for developing a custom work queue.....	704
Define a typecode for a custom work queue.....	706
Define a custom work item type.....	706
Creating a custom work queue class.....	707
Developing the writer for your custom work queue.....	708
Developing workers for your custom work queue.....	709
Bulk insert work queues.....	710
Overview of bulk insert work queues.....	710
Bulk insert work queue declaration and constructor.....	711
Querying for targets of a bulk insert work queue.....	711
Eliminating duplicate items in bulk insert work queue queries.....	712
Processing work items in a custom bulk insert work queue.....	712
Example work queue that bulk inserts its work items.....	712
Examples of custom work queues.....	713
Example work queue that extends WorkQueueBase.....	713
Example work queue for updating entities.....	714
Example work queue with a custom work item type.....	715
Example of the real-time creation of work items.....	717
Delayed processing of real-time work items.....	722
Developing custom batch process.....	724
Custom batch process overview.....	724
Create a custom batch process.....	725
Define a typecode for a custom batch process.....	725
Batch process base class.....	726
Useful properties on class BatchProcessBase.....	726
Useful methods on class BatchProcessBase.....	727
Examples of custom batch processes.....	731
Example batch process for a background task.....	731
Example batch process for unit of work processing.....	732
Working with custom processing.....	734
Categorizing a process typecode.....	735
Updating the work queue configuration.....	735
Implementing IProcessesPlugin	736
Retrying failed batch process items.....	737
Monitoring ClaimCenter processes.....	739
Monitoring ClaimCenter processing using administrative screens.....	739
Monitoring ClaimCenter processes for completion.....	740
Monitoring batch processes by using maintenance tools.....	740
Periodically purging process entities.....	741
Managing user entity updates in batch processing.....	741
Set the external user field in a batch process.....	741

Part 12

Querying and connecting to databases..... 743

38 Query builder APIs.....	745
The processing cycle of a query.....	747
Building a simple query.....	748
Restricting the results of a simple query.....	749
Ordering the results of a simple query.....	750
Accessing the results of a simple query.....	750
Restricting a query with predicates on fields.....	751
Using a comparison predicate with a character field.....	751
Using a comparison predicate with a date and time field.....	756
Using a comparison predicate with a null value.....	758
Using set inclusion and exclusion predicates.....	760
Comparing column values with each other.....	760
Comparing a column value with a literal value.....	761
Comparing a typekey column value with a typekey literal.....	762
Using a comparison predicate with spatial data.....	762
Combining predicates with AND and OR logic	763
Joining a related entity to a query.....	767
Joining an entity to a query with a simple join.....	767
Ways to join a related entity to a query.....	769
Accessing properties of the dependent entity.....	773
Handling duplicates in joins with the foreign key on the right.....	774
Joining to a subtype of an entity type.....	775
Restricting query results by using fields on joined entities.....	776
Restricting query results with fields on primary and joined entities.....	778
Working with row queries.....	780
Selecting columns for row queries.....	781
Applying a database function to a column.....	782
Transforming results of row queries to other types.....	785
Accessing data from virtual properties, arrays, or keys.....	786
Limitations of row queries.....	787
Working with results.....	788
What result objects contain.....	788
Filtering results with standard query filters.....	790
Ordering results.....	794
Useful properties and methods on result objects.....	796
Converting result objects to lists, arrays, collections, and sets.....	799
Updating entity instances in query results.....	800
Testing and optimizing queries.....	801
Performance differences between entity and row queries.....	801
Viewing the SQL select statement for a query.....	802
Enabling context comments in queries on SQL Server.....	803
Including retired entities in query results.....	803
Setting the page size for prefetching query results.....	804
Using settings for case-sensitivity of text comparisons.....	804
Chaining query builder methods.....	806
Working with nested subqueries.....	807
Paths.....	808
Types and methods in the query builder API.....	808
Types and methods for building queries.....	809

Column selection types and methods in the query builder API.....	810
Predicate methods reference.....	813
Aggregate functions in the query builder APIs.....	816
Utility methods in the query builder APIs.....	817
39 Database connection pool.....	819
Reserving a single database connection.....	819
Part 13	
Reference specifications.....	821
40 Integration mapping specification.....	823
Integration mapping files.....	823
Integration mapping file specification.....	824
Integration mapping file combination.....	826
Integration mapping file imports.....	827
Integration mappers.....	828
Integration View filters.....	829
JsonMapper and TransformResult.....	832
Integration mapping examples.....	833
41 Guidewire JSON schema support specification.....	835
JSON schema files.....	835
JSON schema file specification.....	836
JSON schema file combination.....	844
JSON schema imports.....	845
JSON data types and formats.....	847
JSON parsing and validation.....	849
JSON serialization.....	851
JsonObject class.....	852
JSON schema wrapper classes.....	853
XML output and XSD translation.....	855
Releasing schemas.....	859
Externalized JSON schemas.....	859

Support

For assistance, visit the Guidewire Community.

Guidewire customers

<https://community.guidewire.com>

Guidewire partners

<https://partner.guidewire.com>

part 1

Planning integration projects

You can integrate a variety of external systems with ClaimCenter by using services and APIs that link ClaimCenter with custom code and external systems. This overview provides information to help you plan integration projects for your ClaimCenter deployment and provides technical details critical to successful integration efforts.

ClaimCenter addresses the following integration architecture requirements.

A service-oriented architecture

Encapsulate your integration code so that upgrading the core application requires few other changes. Also, a service-oriented architecture enables APIs to use different languages or platforms.

Configurable behavior with the help of external code or external systems

For example, implement special claim validation logic, use a legacy system that generates claim numbers, or query a legacy system.

Sending messages to external systems in a transaction-safe way

Trigger actions after important events happen in ClaimCenter, and notify external systems only if the change is successful and no exceptions occurred. For example, alert a policy database if anyone changes claim information.

Flexible export

Providing different types of export minimizes data conversion logic. Simplifying the conversion logic improves performance and code maintainability for integrating with diverse and complex legacy systems.

Predictable error handling

Find and handle errors cleanly and consistently for a stable integration with custom code and external systems.

Linking business rules to custom task-oriented Gosu or Java code

Let Gosu-based business rules in Guidewire Studio or Gosu templates call Java classes directly from Gosu.

Importing or exporting data to or from external systems

There are multiple ways to import data to and export data from ClaimCenter. You can choose which methods make the most sense for your integration project.

Using clearly defined industry standard protocols for integration points

ClaimCenter provides APIs to retrieve claims, create users, manage documents, trigger events, validate records, and trigger bulk import/export. However, most legacy system integrations require additional integration points customized for each system.

To achieve these goals, the ClaimCenter integration framework provides multiple ways to integrate external code with ClaimCenter.

Web service APIs

Web service APIs are a general-purpose set of application programming interfaces that you can use to query, add, or update Guidewire data, or trigger actions and events programmatically. Because these APIs are web services, you can call them from any language and from any operating system.

A typical use of the web service APIs is to send a new FNOL (First Notice of Loss) to ClaimCenter to set up a new claim.

ClaimCenter supports WS-I web services that use the SOAP protocol. You can use the built-in SOAP APIs, but you can also design your own SOAP APIs in Gosu and expose them for use by remote systems. Additionally, your Gosu code can call web services hosted on other computers to trigger actions or retrieve data.

Plugins

ClaimCenter plugins are classes that ClaimCenter invokes to perform an action or calculate a result. Guidewire recommends writing plugins in Gosu, although you can also write plugins in Java. Gosu code, like Java code, can call third-party Java classes and Java libraries.

Several types of plugins are supported.

Messaging plugins

Send messages to remote systems, and receive acknowledgments of each message. ClaimCenter has a sophisticated transactional messaging system to send information to external systems in a reliable way. Any server in the cluster can create a message for any data change. The only servers that send messages are servers with the messaging server role.

Authentication plugins

Integrate custom authentication systems. For instance, define a user authentication plugin to support a corporate directory that uses the LDAP protocol. Or define a database authentication plugin to support custom authentication between ClaimCenter and its database server.

Document and form plugins

Transfer documents to and from a document management system, and help prepare new documents from templates. Additionally, use Gosu APIs to create and attach documents.

Inbound integration plugins

Multi-threaded inbound integrations for high-performance data import. ClaimCenter provides implementations for reading files and receiving JMS messages, but you can also write your own implementations that leverage the multi-threaded framework.

Other plugins

Some examples are plugins that generate claim numbers (`IClaimNumGenAdapter`) or get a policy related to a claim from a policy administration system (`IPolicySearchAdapter`).

Inbound and outbound file integration

Frameworks for configuring multiple integrations with external systems by using files to transfer data. ClaimCenter uses batch processing to read or create files and workqueues to read or write the records in the files.

Integration Views

Integration Views enables you to select and retrieve data from the ClaimCenter database and to compose and transform that data to meet business needs. Integration Views support the definition of a stable, versioned contract for this data processing. Integration Views also provides a straightforward, declarative way to map underlying ClaimCenter data into a serialized format that conforms to the contract.

GX models

A GX model supports converting the properties of a data type to XML. Supported data types that can be converted include Gosu classes and business data entities, among others. A GX model can include all or a subset of an associated data type's properties. Using GX models to limit the transfer of object data to the minimum subset of required properties conserves resources and improves performance.

REST API framework

A framework that provides the means to define, implement, and publish REST API contracts.

Templates

Generate text-based formats that contain combinations of ClaimCenter data and fixed data. Templates are ideal for name-value pair export, HTML export, text-based form letters, or simple text-based protocols.

Database import from staging tables

Perform a bulk data import into production databases by using temporary loading into “staging” database tables. ClaimCenter performs data consistency checks on data before importing new data, although it does not run validation rules. Database import is faster than adding individual records one at a time. To initially load data from an external system or performing large daily imports from another system, use database staging table import.

The following table compares the main integration methods.

Integration type	Description	What you write
Web services	A full API to manipulate ClaimCenter data and trigger actions externally from any programming language or platform that uses ClaimCenter Web services APIs. Accessed by using the platform-independent SOAP protocol.	<ul style="list-style-type: none"> To publish web services, write Gosu classes that implement each operation as a class method. To consume web services that a Guidewire application publishes, write Java or Gosu code on an external system. The external system calls web services by using the WSDL that ClaimCenter generates. To consume external web services from Gosu, write Gosu code that uses WSDL from the external system. Gosu creates native types from the WSDL that you download to the Studio environment. In all cases, define objects that encapsulate only the data you need to transfer to and from ClaimCenter. The general name for such objects are Data Transfer Objects (DTOs). Define DTOs as Gosu classes or by using XSDs.
Plugins	Classes that ClaimCenter calls to perform tasks or calculate a result. Plugins run in the same Java virtual machine (JVM) as ClaimCenter.	<ul style="list-style-type: none"> Gosu classes that implement a Guidewire-defined interface. Java classes that implement a Guidewire-defined interface. Optionally use OSGi, a Java component system.
Messaging code	Changes to application data trigger events that generate messages to external systems. You can send messages to external systems and track responses called acknowledgments.	<ul style="list-style-type: none"> Gosu code in the Event Fired rule set. These rules create messages (Message objects) that are processed in a separate transaction, possibly on other nodes in the cluster. Messages represent data to send to external systems. Messaging plugin implementations that send the messages to external systems. The most important interface is <code>MessageTransport</code>. There are other optional plugins. Configure one or more messaging destinations in Studio to specify your messaging plugins. After you register your plugins in the Plugins editor, you must also use the Messaging editor for each destination. In all cases, define objects that encapsulate only the data you need to transfer to and from ClaimCenter. The general name for such objects

Integration type	Description	What you write
Inbound and outbound files integration	Inbound and outbound files integrations communicate with external systems by transferring structured data in files. These integrations work with both the local file system and Amazon S3 buckets. ClaimCenter provides frameworks to support these integrations and your code provides the implementation of the data transfer.	<p>is Data Transfer Objects (DTOs). Define DTOs as Gosu classes or by using XSDs.</p> <ul style="list-style-type: none"> • Configuration details. • For inbound files integration, a class implementing the <code>InboundFileHandler</code> interface. • For outbound files integration, a class implementing the <code>OutboundFileHandler</code> interface.
Integration Views	Integration Views provides a versioned contract that maps the ClaimCenter data model into a serialized form that an external system can consume or produce. This contract is decoupled from the data model so that changes to either the contract or the data model do not interfere with each other.	<ul style="list-style-type: none"> • Mappings that transform an object of a particular type first into an intermediate object that conforms to a specified schema and then to a JSON object or XML element. • The schema that defines the structure of the data involved in the custom processes. • Optionally, filters specifying the parts of the schema to use as output or for presentation.
GX models	<p>GX models support the conversion of properties of an entity or other data type to XML. The GX model editor in Studio defines the data type's properties to include in the GX model. As you define the model, ClaimCenter generates an XSD schema definition.</p> <p>Subsequently, an instance of the GX model containing XML data can be used to integrate with external systems. For example, messaging plugins could use a GX model to send XML to external systems. Also, a web service could accept the XML data contained in a GX model as a payload from an external system.</p>	<ul style="list-style-type: none"> • Using the GX Model editor in Studio, you customize the properties in an entity or other data type to export in XML format. • Integration code that uses the GX model.
REST API framework	<p>The Guidewire REST API framework, in combination with Guidewire Integration Views, provides ClaimCenter with support for the following:</p> <ul style="list-style-type: none"> • Swagger and OpenAPI 2.0 API schemas • JSON data schema payloads • Mappings from the ClaimCenter data model schema to JSON 	<ul style="list-style-type: none"> • Swagger and JSON schema files • Mapping files • Classes that perform the actual API work • API handler classes
Templates	<p>Several data extraction mechanisms that perform database queries and format the data as necessary. For example:</p> <ul style="list-style-type: none"> • Sending notifications as form letters and use plain text with embedded Gosu code to simplify deployment and ongoing updates. • Designing a template that exports HTML for development of web-based reports of ClaimCenter data. 	<ul style="list-style-type: none"> • Text files that contain small amounts of Gosu code.
Database import	You can import large amounts of data into ClaimCenter by first populating a separate set of staging database tables. Staging tables are	<ul style="list-style-type: none"> • Custom tools that take legacy data and convert them into database tables of data.

Integration type	Description	What you write
	temporary versions of the data that exist separately from the production database tables. Next, use ClaimCenter web service APIs or command line tools to validate and import that data.	
Links	Link from a ClaimCenter screen to a URL address, such as a screen in another InsuranceSuite application or an external application. The link is implemented by using an ExitPoint PCF file.	<ul style="list-style-type: none">• An ExitPoint PCF file.• Incorporate the ExitPoint into the originating ClaimCenter screen.

Overview of integration components

The Integration Guide is the main source for integration information. Other useful information can be found in the various API References and the Data Dictionary.

[Java API reference](#)

The *Java API Reference* includes the specification of the plugin definitions for Java plugin interfaces, entity types, typelist types, and other types available from Java.

The Reference contents are static. The script that regenerates the Java API (`gwb genJavaApi`) does not regenerate the Reference. Therefore, your own data model changes are not reflected in the Reference documentation. However, your changes to entity types, typecodes, and new properties are available from Java code in your Java IDE.

[Web service API reference documentation](#)

On a running ClaimCenter server, you can get up-to-date WSDL from published services.

For WS-I web services, there is no built-in Javadoc-style generation. The exact method signatures and syntax vary based on the language which the SOAP implementation that generates libraries from the WSDL.

[See also](#)

- “Generating and publishing WSDL” on page 80

[Gosu API reference](#)

The *Gosu API Reference* is a browser-based listing of Gosu classes, methods, and data. The Reference is particularly valuable for programmers implementing Gosu plugins.

The *Gosu API Reference* can be generated from the command line by using the `gwb gosudoc` tool.

[Data Dictionary documentation](#)

The *ClaimCenter Data Dictionary* provides documentation on classes that correspond to ClaimCenter data. You must generate the *Data Dictionary* before using it.

The *ClaimCenter Data Dictionary* typically has more information about data-related objects than the various API References have for the same class/entity. The *Data Dictionary* documents only classes corresponding to data defined in the data model. It does not document any API functions or utility classes.

Preparing for integration development

During integration development, you edit configuration files in the hierarchy of files in the product installation directory. In most cases, you modify data only through the Guidewire Studio interface, which handles any SCM

(Source Control Management) requests. The Studio interface also copies read-only files to the configuration module in the file hierarchy for your files and makes files writable as appropriate.

However, in some cases you need to add files directly to directories in the configuration module hierarchy, such as Java class files for Java plugin support. The configuration module hierarchy for your files is in the hierarchy:

```
ClaimCenter/modules/configuration
```

Some of the main configuration subdirectories are described in the following table.

Directory under the configuration module	Files that the directory contains
config/iso	Your Insurance Services Office (ISO) files.
config/metro	Your Metropolitan (police report/inquiry) files.
config/web	Your web application configuration files, also known as PCF files.
config/logging	The logging configuration file log4j2.xml.
config/templates	Two types of templates relevant for integration.

Plugin templates

Use Gosu plugin templates for a small number of plugin interfaces that require them. These plugin templates extract important properties from entity instances and generate text that describes the results. Whenever ClaimCenter needs to call the plugin, ClaimCenter passes the template results to the plugin as String parameters.

Messaging templates

Use optional Gosu messaging templates for your messaging code. Use messaging templates to define notification letters or other similar messages contain large amounts of text but a small amount of Gosu code.

The base configuration provides versions of some of these templates.

plugins	<p>Your Java plugin files.</p> <p>Register your plugin implementations in the Plugins registry in Studio. When you register the plugin in the Plugins registry, you can specify a <i>plugin directory</i>, which is the name of a subdirectory of the plugins directory. If you do not specify a subdirectory, ClaimCenter uses the shared subdirectory as the plugin directory.</p> <p>For a messaging plugin, you must register this information in two different registries:</p> <ul style="list-style-type: none"> • The plugin registry in the plugin editor in Studio • The messaging registry in the Messaging editor in Studio.
---------	---

Some additional important integration-related directories are described in the following table.

Directory under the ClaimCenter installation directory	Files that the directory contains
ClaimCenter installation directory	<p>Command-prompt tools such as gwb.bat. Use for the following integration tasks:</p> <ul style="list-style-type: none"> • Regenerating the Java API libraries and local WSI web service WSDL. • Regenerating the <i>Data Dictionary</i>.
modules/configuration/gsrc/wsi/ local/gw/webservice/	WSDL files generated locally.
modules/configuration/gsrc/wsi/ local/gw/wsi/	
admin	Command-prompt tools that control a running ClaimCenter server. Almost all of these tools are small Gosu scripts that call public web service APIs.

Regenerating integration libraries and WSDL

You must regenerate the Java API libraries and SOAP WSDL after you make certain changes to the product configuration. Regenerate these files in the following situations.

- After you install a new ClaimCenter release.
- After you make changes to the ClaimCenter data model, such as data model extensions, typelists, field validators, and abstract data types.

As you work on both configuration and integration tasks, you may need to regenerate the Java API libraries and SOAP WSDL frequently. However, if you make significant configuration changes and then work on integration at a later stage, wait until you need the APIs updated before regenerating ClaimCenter files.

Regenerating the Java API libraries

For Java development, generate the Java entity libraries with the following command.

```
gwb genJavaApi
```

As part of its normal behavior, the script displays some warnings and errors. Note that the Java API libraries are regenerated, but the command does not regenerate the static *Java API Reference* documentation.

The location for the Java generated libraries is:

```
ClaimCenter/java-api/lib
```

WS-I web service regeneration of WSDL for local GUnit tests

For web service development, you can write unit tests that call web services on the same computer. Generate WSDL for testing only using the following command.

```
gwb genWsiLocal
```

As part of its normal behavior, the script displays some warnings and errors.

The location for WS-I web service WSDL that you can use for writing test classes is:

```
ClaimCenter/modules/configuration/gsrc/wsi/local/
```

Required files for integration programmers

Depending on what kind of integrations you require, there are special files you must use. Examples are libraries to compile your Java code against, to import into a project, or to use in other ways.

The following list shows several types of integration files that you use in integrations with ClaimCenter and examples of what those files represent.

Web services

Files that you can use with SOAP API client code.

Plugin interfaces

Plugin interfaces for your code to respond to requests from ClaimCenter to calculate a value or perform a task.

Java API libraries

Entity types are defined in the data model for ClaimCenter. These definitions have built-in properties and can also have data model extension properties. For example, *Claim* and *Address* are ClaimCenter entities. To access entity data and methods from Java, you need to use Java API libraries.

In all cases, ClaimCenter entities such as *Claim* contain data properties that can be manipulated either directly or from some contexts by using getters and setters (*get...* and *set...* methods).

Depending on the type of integration point, there might be additional methods available on the objects. These additional domain methods often provide valuable functionality.

Entity access	Description	Entities	Necessary libraries
Gosu plugin implementation	Plugin interface defined in Gosu.	Full entity instances	None
Java plugin implementation, including optional OSGi support	Java code that accesses an entity associated with a plugin interface parameter or return value.	Full entity instances	Java API libraries
Java class called from Gosu	Java code called from Gosu that accesses an entity passed as a parameter from Gosu, or a return result to be passed back to Gosu.	Full entity instances	Java API libraries
Web services	WS-I web services	No support for entity types as n/a arguments or return values. Use data transfer objects (DTOs) instead.	

For web services, the DTOs that you use can be any of the following:

- Gosu class instances that contain only the properties required for the integration point. You need to declare the class `final` and add the annotation `gw.xml.ws.annotation.WsiExportable`.
- XML objects with their structure defined in XSD files.
- XML objects with their structure defined with the GX modeler. The GX modeler is a tool to generate an XML model with only the desired subset of properties for each entity for each integration point.

Public IDs and integration code

ClaimCenter creates its own unique ID for every entity in the system after it fully loads in the ClaimCenter system. However, this internal ID cannot be known to an external system while the external system prepares its data. Consequently, if you get or set ClaimCenter information, use unique public ID values to identify an entity from external systems connecting to a Guidewire application.

Your external systems can create this public ID based on its own internal unique identifier, based on an incrementing counter, or based on any system that can guarantee unique IDs. Each entity type must have unique public IDs within its class. For instance, two different `Address` objects cannot have the same public ID.

However, a claim and an exposure may share the same public ID because they are different entities.

If loading two related objects, the incoming request must tell ClaimCenter that they are related. However, the web service client does not know the internal ClaimCenter IDs as it prepares its request. Creating your own public IDs guarantees the web service client can explain all relationships between objects. This is true particularly if entities have complex relationships or if some of the objects already exist in the database.

Additionally, an external system can tell ClaimCenter about changes to an object even though the external system might not know the internal ID that ClaimCenter assigned to it. For example, if the external system wants to change a contact's phone number, the external system only needs to specify the public ID of the contact record.

ClaimCenter allows most objects associated with data to be tagged with a public ID. Specifically, all objects in the *Data Dictionary* that show the `keyable` attribute contain a public ID property. If your API client code does not need particular public IDs, let ClaimCenter generate public IDs by leaving the property blank. However, other non-API import mechanisms require you to define an explicit public ID, for instance database table record import.

If you choose not to define the public ID property explicitly during initial API import, later you can query ClaimCenter with other information. For example, you could pass a contact person's full name or taxpayer ID if you need to find its entity programmatically.

You can specify a new public ID for an object. From Gosu, set the `PublicID` property.

Creating your own public IDs

Suppose a company called ABC has two external systems, each of which contains a record with an internal ID of 2224. Each system generates public ID by using the format "{company}:{system}:{recordID}" to create unique public ID strings such as "abc:s1:2224" and "abc:s2:2224".

To request ClaimCenter automatically create a public ID for you rather than defining it explicitly, set the public ID to the empty string or to `null`. If a new entity's public ID is blank or `null`, ClaimCenter generates a public ID. The ID is a two-character ID, followed by a colon, followed by a server-created number. For example, "cc:1234". Guidewire reserves for itself all public IDs that start with a two-character ID and then a colon.

Public IDs that you create must never conflict with ClaimCenter-created public IDs. If your external system generates public IDs, you must use a naming convention that prevents conflict with Guidewire-reserved IDs and public IDs created by other external systems.

The prefix for auto-created public IDs is configurable using the `PublicIDPrefix` configuration parameter. If you change this setting, all explicitly-assigned public IDs must not conflict with the namespace of that prefix.

The maximum public ID length is 64 characters.

Important – Integration code must never set a public IDs to a `String` that starts with a two-character ID and then a colon. Guidewire strictly reserves all such IDs. If you use the `PublicIDPrefix` configuration parameter, integration code that sets explicit public IDs also must not conflict with that namespace. Additionally, plan your public ID naming to support large (long) record numbers. Your system must support a significant number of records over time and stay within the public ID length limit.

Guidewire internal classes and methods

Some code packages contain Guidewire internal classes that are reserved for Guidewire use only. Gosu code written to configure ClaimCenter must never use an internal class nor call any method of an internal class for any reason. Future releases of ClaimCenter can change or remove an internal class without notification.

The following packages contain Guidewire internal classes.

- All packages in `com.guidewire.*`
- Any package whose name or location includes the word `internal`

Gosu configuration code can safely use classes defined in any `gw.*` package, except for those packages whose name or location includes the word `internal`.

Some Gosu classes are visible in Studio, but are not intended for use. You can distinguish these Gosu classes because they have no visibility annotations (neither `@Export` nor `@ReadOnly`) and they are not in a `gw` package. Do not use these methods in configuration. The methods are unsupported and may change or be withdrawn without notice.

chapter 2

Proxy servers

Guidewire recommends deploying proxy servers to insulate ClaimCenter from the external Internet. Because incoming requests are the most dangerous, this is most important for integrating ClaimCenter with Insurance Service Office (ISO) because ISO must initiate network requests directly to ClaimCenter. However, Guidewire recommends proxy servers for outgoing requests to Metropolitan Reporting Bureau and outgoing geocoding requests.

Several ClaimCenter integration options require outgoing messages, including ISO integration, geocoding integration, and Metropolitan integration. In addition, ISO integration requires incoming requests from the Internet to ClaimCenter. Placing a proxy server between the external Internet and ClaimCenter insulates ClaimCenter from some types of attacks and partitions all network access for maximum security.

Additionally, some of the integration points require encrypted communication. Because encryption in Java tends to be lower performance than in native code that is part of a web server, encryption can be off-loaded to the proxy server. For example, instead of the ClaimCenter server directly encrypting HTTPS/SSL connections to an outsider server, ClaimCenter can contact a proxy server with standard HTTP requests. Standard requests are less resource intensive than SSL encrypted requests. The proxy server running fast compiled code connects to the outside service using HTTPS/SSL.

A proxy server that handles incoming connections from an external Internet service to ClaimCenter and not just outgoing requests from ClaimCenter is sometimes called a reverse proxy server. For the sake of simplicity, this topic refers to any server that handles incoming requests as a proxy server. Your server might handle only outgoing requests if you do not need to intercept incoming requests.

Resources for understanding and implementing SSL

Some proxy server configurations use SSL encryption. Encryption concepts and proxy configuration details are complex, and full documentation on this process is outside the scope of ClaimCenter documentation.

For more information about SSL encryption and Apache-specific documentation related to SSL, refer to all of the following resources.

Encryption-related documentation	For more information, see this location
High-level overview of public key encryption	http://en.wikipedia.org/wiki/Public_key
Detailed description of public key encryption	ftp://ftp.pgp.org/pub/pgp/7.0/docs/english/IntroToCrypto.pdf
Detailed description of SSL/TLS Encryption	http://httpd.apache.org/docs/current/ssl/ssl_intro.html
Overview of Apache's SSL module	http://httpd.apache.org/docs/current/mod/mod_ssl.html

Encryption-related documentation	For more information, see this location
Overview of Apache's proxy server module	http://httpd.apache.org/docs/current/mod/mod_proxy.html

Web services and proxy servers

If your ClaimCenter deployment must call out to web services hosted by other computers, for maximum security always connect to it through a proxy server.

You can vary the URL to remote-hosted web services based on configuration environment settings on your server, specifically the `env` and `serverid` settings. For example, if running in a development environment, directly connect to the remote service or through a testing-only proxy server. In contrast, if running in the production environment, always connect through the official proxy server.

See also

- For ISO, see “Insurance services office (ISO) integration” on page 579.
- For Metropolitan, see “Metropolitan Reporting Bureau integration” on page 617.
- For geocoding, see “Geographic data integration” on page 249.
- For configuring web services URLs to support proxy servers, see the *Configuration Guide*.

Configuring a proxy server with Apache HTTP Server

Apache HTTP Server is a popular open source web server that can be configured as a proxy server. This section is intended only if you need to use the ISO Apache HTTP Server examples included with ClaimCenter. Also use this section to integrate the relevant security elements into a current Apache configuration for an existing Apache proxy server.

This section presents the generic Apache HTTP Server configuration, and the next section describes the different proxy building blocks. You can add one or more building blocks to your own Apache configuration file as appropriate.

Install Apache HTTP Server: Basic checklist

About this task

This section describes the high-level Apache installation and security instructions. A full detailed set of Apache instructions is outside the scope of this Guidewire documentation.

Procedure

1. Download Apache HTTP server. Get it from <http://httpd.apache.org>.

The Apache configuration files blocks listed in this topic were designed for Apache 2.2.X. Guidewire has observed problems with Apache HTTP versions older than version 2.2.3. Do not use older versions. To use these examples, use the latest 2.2.X release and confirm X is a number 3 or greater.

2. Install Apache HTTP server.
3. Download and install the SSL security Apache module.
4. Install Apache HTTP server as a background UNIX daemon or Windows service.
5. Configure the Apache directive configuration file.
6. Make appropriate changes to your firewall. If you already have some sort of corporate firewall, you must make holes in your firewall for all integration points.
7. Install any necessary SSL certificates and SSL keys.
8. Enable Apache modules. Enable the following Apache modules `mod_proxy` (proxies in general), `mod_proxy_http` (HTTP proxies), `mod_proxy_connect` (SSL tunneling), `mod_ssl` (SSL encryption).

Certificates, private keys, and passphrase scripts

Security file	Description
\$DestinationTrustedCACertFile	File containing the certificate used to sign the destination web site.
\$ReverseProxyTrustedCertFile	File containing the certificate for the reverse proxy site. To ensure that the certificate is recognized by source systems, ensure a Trusted Certification Authority signs it.
\$ReverseProxyTrustedProtectedPrivateKeyFile	File containing the private key used to decrypt the messages in the source to reverse proxy communication. This file is generally signed by a passphrase script \$ReverseProxyTrustedPassPhraseScript.

The \$ReverseProxyTrustedProtectedPrivateKeyFile is very sensitive. If it is exposed, it may allow an elaborate attacker to impersonate your web site by coupling this exploit with DNS corruptions. Therefore, this private key must be secured by all means.

Rather than displaying that private key in a file, it is a common practice to secure that private key through a pass-phrase. The DMZ proxy would then be provided with both the protected private key file and with a script that would return the pass-phrase under specific security conditions. The logic of the script and the conditions for returning the right pass-phrase are the secured DMZ proxy's administrator responsibility. The script's goal is to prevent the pass-phrase to be returned if not called from the right proxy instance and from a non-corrupted environment.

Proxy server integration types

ISO proxy communication

ClaimCenter and ISO exchange different types of messages. ClaimCenter sends messages 1 and 2 to ISO. For these messages, use the configuration for the downstream proxy. ISO sends Message 3 asynchronously to ClaimCenter. For these message, use the configuration for the upstream proxy for service connections.

See also

- “Downstream proxy with encryption” on page 41
- “Upstream (reverse) proxy with encryption for service connections” on page 41

Metropolitan proxy communication

For Metropolitan reports, use the downstream proxy configuration to connect to the Metropolitan Request URL. Use the same URL later to poll for availability of results.

ClaimCenter retrieves the report using the separate URL called the Metropolitan Report Retrieval URL. Use the same configuration block, but with the different URL.

See also

- “Downstream proxy with encryption” on page 41

Bing maps geocoding service communication

The geocoding plugin only initiates communications with geocoding services. It never responds to communications initiated from the external Internet. Therefore, you do not need a reverse proxy server to insulate the geocoding plugin and ClaimCenter from incoming Internet requests. However, Guidewire recommends insulating outgoing geocoding requests through a proxy server as a best practice.

Configure the Bing maps geocoding plugin to use a proxy server

1. In Guidewire Studio, configure the web service collection for the Bing Maps web service.
 - a. Navigate to the web service collection at:
Resource > Classes > wsi > remote > microsoft > bingmaps
2. In the Web Service Collection editor, click **Add Setting** and then **Override URL**.
3. In the **Override URL** field, enter your proxy server URL and port number for the Bing Maps web service.
4. In the configuration of your Apache proxy server, add a configuration building block for the Bing Maps URL. Follow the pattern for downstream proxy with no encryption configuration building blocks.

If you use Guidewire ContactManager and geocoding is enabled within ContactManager, typically you can use the same configuration building block for communication between ContactManager and Bing Maps. Allow connections from the IP addresses of ClaimCenter and ContactManager in each configuration building block. Also, configure the web service collection for Bing Maps in ContactManager Studio with the overriding URL of your proxy server.

See also

- “Downstream proxy with no encryption” on page 40

SSL encryption for users

You may want to use the Apache server to handle SSL encryption from users to ClaimCenter and reduce the processing burden of SSL encryption in Java on the ClaimCenter server. Use the upstream or reverse proxy Apache configuration building block.

See also

- “Upstream (reverse) proxy with encryption for user connections” on page 42
- For information about managing secure communications, see the *Administration Guide*.

Proxy building blocks

The following subsections list building blocks of configuration text for Apache configuration files. For each building block, you must substitute all values that are prepended by dollar signs (\$). Actual Apache configuration files must not have the dollar sign in the actual file.

For instance, locate the following configuration line.

```
Listen $PROXY_PORT_NUMBER_HERE
```

The dollar sign (\$) appears in configuration building blocks to indicate values that must be substituted with hard-coded values. Replace all these values and leave no “\$” characters in the final file.

Replace the line with the following configuration setting to listen on port 1234.

```
Listen 1234
```

Downstream proxy with no encryption

In this configuration, a source system calls the proxy, which transmits the request unchanged to the destination URL. The reply follows the opposite path unencrypted.

Use the following Apache configuration building block.

```
#Disable forward proxying for security purposes
ProxyRequests Off

#The reverse proxy listens to the source system on the reverse proxy port.
Listen $PROXY_PORT_NUMBER_HERE
```

```

<VirtualHost *:$PROXY_PORT_NUMBER_HERE>
<Proxy *>
    Order Deny,Allow
    Deny from all

    # The Virtual Host accepts requests only from the source system
    Allow from $SourceSystem
</Proxy>

    # The Virtual Hosts associates the packet to the destination URL
    ProxyPass $SOURCE_URL $DESTINATION_URL

    #Logs redirected to appropriate location
    ErrorLog $ApacheErrorLog

</VirtualHost>

```

Replace the \$SOURCE_URL value with the source URL. To redirect all HTTP traffic on all URLs on the source IP address and port, use the string “/”, which is just the forward slash, with no quotes around it.

Replace the \$DESTINATION_URL value with the destination domain name or URL.

Downstream proxy with encryption

In this configuration, a source system calls the proxy, which transmits the request to the destination URL. The reply follows the opposite path. The proxy to destination system communication is encrypted for the request and also for the reply.

Use the following Apache configuration building block.

```

#SSL sessions are cached to ensure possible reuse across sessions
SSLSessionCache shm:$SSL_CACHE(512000)
SSLSessionCacheTimeout 300

#Disable forward proxying for security purposes
ProxyRequests Off

#The reverse proxy listens to the source system on the reverse proxy port.
Listen $PROXY_PORT_NUMBER_HERE

<VirtualHost *:$PROXY_PORT_NUMBER_HERE>
<Proxy *>
    Order Deny,Allow
    Deny from all

    # The Virtual Host accepts requests only from the source system
    Allow from $SourceSystem
</Proxy>

    # The Virtual Hosts associates the packet to the destination URL
    ProxyPass / $DestinationURL

    #Communication is encrypted on the reverse proxy to destination system leg
    SSLProxyEngine on

    #The Reverse proxy checks the destination's certificate
    #using the appropriate Trusted CA's certificate
    SSLProxyCACertificateFile $DestinationTrustedCACertFile

    #Logs redirected to appropriate location
    ErrorLog $ApacheErrorLog

</VirtualHost>

```

Upstream (reverse) proxy with encryption for service connections

In this configuration, a source system calls the reverse proxy, which transmits the request to the destination URL. The reply follows the opposite path. The source system to reverse proxy communication is encrypted for both request and reply.

Use the following Apache configuration building block.

```

#SSL sessions are cached to ensure possible reuse across sessions
SSLSessionCache shm:$SSL_CACHE(512000)
SSLSessionCacheTimeout 300

#Private keys are secured through a pass-phrase

```

```

SSLPassPhraseDialog exec:$ReverseProxyTrustedPassPhraseScript

#Disable forward proxying for security purposes
ProxyRequests Off

#The reverse proxy listens to the source system on the reverse proxy port.
Listen $REVERSEPROXY_PORT_NUMBER_HERE

<VirtualHost *:$REVERSEPROXY_PORT_NUMBER_HERE>
<Proxy *>
    Order Deny,Allow
    Deny from all

    # The Virtual Host accepts requests only from the source system
    Allow from $SourceSystem
</Proxy>

# The Virtual Host associates the packet to the destination URL
ProxyPass / $DestinationURL

#Communication is encrypted on the source system to reverse proxy leg
SSLEngine on

#The Virtual Host authenticates to the source system providing its certificate
SSLCertificateFile $ReverseProxyTrustedCertFile

#The communication security is achieved using the PrivateKey, which is secured
#through a pass-phrase script.
SSLCertificateKeyFile $ReverseProxyTrustedProtectedPrivateKeyFile

#Logs redirected to appropriate location
ErrorLog $ApacheErrorLog

</VirtualHost>

```

Upstream (reverse) proxy with encryption for user connections

Use the Apache server to handle SSL encryption from users to ClaimCenter and thus reduce the processing burden of SSL encryption in Java on the ClaimCenter server. Use the following configuration building block.

```

#Encrypted Reverse Proxy
<VirtualHost *:portnumber>

#Allow from the authorized remote sites only
<Proxy *>
    Order Deny,Allow
    Allow from all
</Proxy>

# Access to the root directory of the application server is not allowed
<Directory />
    Order Deny,Allow
    Deny from all
</Directory>

#Access is allowed to the cc directory and its subdirectories for the authorized sites only
<Directory /cc>
    Order Deny,Allow
    Allow from all

# Never allow communications to be not encrypted
SSLRequireSSL

#The Cipher strength must be 128 (maximal cipher size authorized
#all communication secured
SSLRequire %{SSL_CIPHER_USEKEYSIZE} >= 128 and %{HTTPS} eq "true"
</Directory>

#Classic command to take into account an Internet Explorer issue
SetEnvIf User-Agent ".*MSIE.*" \
nokeepalive ssl-unclean-shutdown \
 downgrade-1.0 force-response-1.0

#Encryption secures the Internet to Encrypted Reverse Proxy communication
#Listing of available encryption levels available to Apache
SSLEngine on
SSLCipherSuite ALL:!ADH:!EXPORT56:RC4+RSA:+HIGH:+MEDIUM:+LOW:+SSLv2:+EXP:+eNULL

#The Virtual Host authenticates to the user providing its certificate
SSLCertificateFile conf/<certificate_filename>.crt

#The communication security is achieved using the PrivateKey, which is secured through
#a pass-phrase script.
SSLCertificateKeyFile conf/<certificate_filename>-secured.pem

#The Virtual Host associates the request to the internal Guidewire product instance

```

```
ProxyPass      /<product>400 <url of the product server>
ProxyPassReverse    /<product>400 <url of the product server>

#Logs redirected to appropriate location
ErrorLog      logs/encrypted_<product>.log

</VirtualHost>
```

Modify the server to receive incoming SSL requests

To enable ClaimCenter to respond to a request over SSL from a particular inbound connection, your proxy handles encryption. The connection between ClaimCenter and the proxy server remains unencrypted. Configure the proxy to know the URL and port (location) of the server that originates the request.

Procedure

1. Edit your proxy server configuration so it is aware of the following items.
 - The externally-visible domain name of the reverse proxy server
 - The port number of the reverse proxy server
 - The protocol the client used to access the proxy server, in this case HTTPS
2. To ensure your ClaimCenter server is aware of the proxy, edit the web application container server configuration `CATALINA_HOME/conf/server.xml` on your ClaimCenter server. Add another connector as shown in the following XML snippet.

```
<!-- Define a non-SSL HTTP/1.1 Connector on port <port number> to receive decrypted
     communication from Apache reverse proxy on port 11410 -->
<Connector acceptCount="100" connectionTimeout="20000" disableUploadTimeout="true"
            enableLookups="false" maxHttpHeaderSize="8192" maxSpareThreads="75" maxThreads="150"
            minSpareThreads="25" port="portnumber" redirectPort="8443" scheme="https" proxyName="hostname"
            proxyPort="portnumber">
</Connector>
```

3. You must substitute the following parameters contained in the snippet.

port The port number for the additional connector for access through the proxy

proxyName The deployment server's name

proxyPort The port for encrypted access through Apache

scheme The protocol used by the client to access the server

4. After configuring the `server.xml` file, restart your application server.

Java and OSGi support

ClaimCenter supports the deployment of Java code. There are multiple reasons to write and deploy Java code in ClaimCenter, including accessing entity data. For example, you can implement ClaimCenter plugin interfaces by using a Java class instead of a Gosu class. If you implement a plugin interface in Java, optionally you can write your plugin implementation as an OSGi bundle. The OSGi framework is a Java module system and service platform that supports cleanly isolating code modules and any necessary Java libraries. Guidewire recommends OSGi for all new Java plugin development. You can also write Java code that you can call from any Gosu code in ClaimCenter, such as from rule sets or other Gosu classes.

In all cases for Java development, you must use an IDE for Java development separate from Guidewire Studio. ClaimCenter does not support adding or modifying Java class files through Guidewire Studio. Although Studio does not hide user interface tools that add Java classes to the file hierarchies, Guidewire does not support coding in Java in Studio.

Accessing Gosu types from Java

From Gosu, you can call Java types, including added third-party Java classes and libraries. However, from Java you cannot access Gosu types without using a language feature called reflection. *Reflection* means to ask the type system at run time about types. Using reflection to access Gosu types means that the editor cannot flag missing or misspelled method or property names at compile time.

You must use reflection to access in Java the following Gosu language elements.

- Gosu classes
- Gosu interfaces
- Gosu enhancements

See also

- “Using reflection to access Gosu classes from Java” on page 51
- *Gosu Reference Guide*

Implementing plugin interfaces in Java and optionally OSGi

A typical use of Java code is to implement plugin interfaces to ClaimCenter.

The main steps to implement a plugin in Java are described below.

1. Regenerate the Java libraries.

2. Write a class that implements the plugin interface.
3. Deploy your Java classes and any dependent libraries. The deployment details vary based on whether you use OSGi to encapsulate your Java code.
4. Register your plugin implementation in Guidewire Studio.

Choosing between a Java plugin and an OSGi plugin

If you implement a plugin interface in Java, there are two ways to deploy your code.

- Java plugin – A Java class. You must use an IDE other than Studio. If you write your plugin in Java, you must regularly regenerate the Java API libraries to compile against the latest libraries. You can use any Java IDE. You can choose to use the included IntelliJ IDEA with OSGi Editor for Java plugin development even if you do not choose to use OSGi.
- OSGi plugin – A Java class encapsulated in an *OSGi* bundle. You must use an IDE other than Studio. The OSGi framework is a Java module system and service platform that helps cleanly isolate code modules and any necessary Java libraries. To simplify OSGi configuration, ClaimCenter includes an application called IntelliJ IDEA with OSGi Editor.

Guidewire recommends OSGi for all new Java plugin development. ClaimCenter supports OSGi bundles only to implement a ClaimCenter plugin interface and any of your related third-party libraries. It is unsupported to install OSGi bundles for any other purpose.

The most important benefits of OSGi for ClaimCenter plugin development are described below.

- Safe encapsulation of third-party Java JAR files. OSGi loads types in a way that reduces compatibility problems between OSGi bundles, or between an OSGi component and libraries that ClaimCenter uses. For example, dependencies on specific third-party packages and classes are explicit in manifest files and validated at server start-up.
- Dependency injection support using declarative services. Declarative services use Java annotations and interfaces to declare dependencies between your code and other APIs. For example, your code does not declare that it needs a specific class for a task. Instead, your code uses Java interfaces to define which services it needs. The OSGi framework ensures the appropriate API or objects are available. The OSGi framework tracks dependencies and handles instantiates objects as needed. For example, your OSGi plugin might depend on a third-party OSGi library that provides a service. Your plugin code can use declarative services to access the service.

Beware of a terminology issue in Guidewire documentation with the word “bundle.”

- In nearly all Guidewire documentation, “bundle” refers to a programmatic abstraction of a database transaction and a set of database rows to update.
- In the OSGi standard, “bundle” refers to a registered OSGi component. ClaimCenter documentation relating to Java and plugins sometimes refers to “OSGi bundles.”

See also

- *Gosu Reference Guide*

Your IDE options for plugin development in Java

To deploy Java code in ClaimCenter, you can use any Java IDE that is separate from Guidewire Studio. Guidewire does not support the editing of Java code directly in Studio.

To write an OSGi plugin implementation, you have several IDE options.

- IntelliJ IDEA with OSGi Editor (included with ClaimCenter) – A specially-configured instance of IntelliJ IDEA and included with ClaimCenter. This is a different application from Studio. This IntelliJ IDEA instance includes a special IntelliJ IDEA plugin for OSGi plugin configuration. For OSGi plugin development, Guidewire recommends using IntelliJ IDEA with OSGi Editor. The included plugin editor configures OSGi configuration files such as the bundle manifest.

- Other Java IDEs, such as Eclipse – You can use other Java IDEs such as Eclipse or your own version of IntelliJ IDEA. However, you must manually configure OSGi files and bundle manifest manually according to the OSGi standard.

The following table summarizes options for development environments for plugin development in Java.

IDE	Gosu plugin	Java plugin (no OSGi)	OSGi plugin (Java with OSGi)
IntelliJ IDEA with OSGi Editor (included with ClaimCenter)	--	Yes	Yes
Eclipse or other Java IDE of your own choice	--	Yes	Yes, though requires manual configuration of OSGi files and bundle manifest.
Guidewire Studio	Yes	--	--

Java IDE inspections that flag unsupported internal APIs

The Java API allows you to use the same Java types that you can use in Gosu. However, Guidewire specifies some methods and fields on these types for internal use only. Do not use any of these *internal APIs*. Guidewire indicates internal API methods and properties with the annotation `@gw.lang.InternalAPI`.

In Gosu, methods and fields with that annotation are hidden. Gosu code that uses internal APIs triggers compilation errors.

In Java, when you are using your own IDE separate from Studio, internal APIs are visible although unsupported. Depending on what IDE you use, you might require additional configuration of your IDE. The following table summarizes the options for internal API code inspections.

IDE	Java IDE Includes Internal API Inspection
IntelliJ IDEA with OSGi Editor (included with ClaimCenter)	Yes
Separate IntelliJ IDEA instance that you provide	Optional installation. For compatibility with specific IntelliJ IDEA versions, please contact Guidewire Customer Support.
Eclipse or other Java IDE of your own choice	No. There is no equivalent code inspection available. If you use another IDE and you are unsure of the status of a particular method or field, navigate to it and see if the declaration has the annotation <code>@gw.lang.InternalAPI</code> .
Guidewire Studio	IMPORTANT: Guidewire Studio does not support Java coding directly in the IDE. Do not create Java classes directly in Studio. If you want to code in Java, you must use a separate IDE for Java development.

Install the internal API inspection in IntelliJ IDEA

As an alternative to the recommended approach for Java development, you can use internal API inspection in a separate instance of IntelliJ IDEA.

About this task

Guidewire recommends using the included IntelliJ IDEA with OSGi Editor for Java development.

The IntelliJ IDEA with OSGi Editor includes the code inspection for the `@InternalAPI` annotation. If you use this included application, you do not need to install any code inspections to flag internal API usage.

If you choose to use your own separate instance of IntelliJ IDEA, you might be able to use the Internal API inspection, depending on your version of IntelliJ IDEA. For compatibility with specific IntelliJ IDEA versions, contact Guidewire Customer Support.

Procedure

1. Open your separate instance of IntelliJ IDEA.
2. Navigate to **File > Settings > Plugins**.
3. On the top tab bar, in the **Manage Repositories, Configure Proxy, or Install Plugin from Disk**  menu, click **Install Plugin from Disk**.
4. Navigate to the following directory.
`ClaimCenter/studio/pluginstudio-plugins/internal-api-idea-plugin/lib/`
5. Select the JAR file in that directory.
6. In IntelliJ IDEA, click **Restart IDE**.
7. Navigate to **File > Settings > Editor > Inspections**.
8. Expand the node **Portability**.
9. Select the check box **Use of internal API**.
10. Click **OK**.

Accessing entity data from Java

An entity type is an abstract representation of Guidewire business data of a certain type. Define entity types in ClaimCenter data model configuration files. Entity types have built-in properties and you can optionally add extension properties. **Claim** and **Address** are examples of entity types.

You can write Java code that accesses entity data from the following components.

- Java plugin implementations
- OSGi plugin implementations written in Java
- Other Java classes called from Gosu

After you call the script that regenerates ClaimCenter Java API libraries, use the libraries to write Java code that uses entity data.

Using a separate Java-capable IDE, add the Java API library directories as a dependency in your project in the separate IDE. Compile your Java code against these libraries.

Do not create Java classes directly in Studio. To code in Java, you must use a separate IDE for Java development. For example, use IntelliJ IDEA with OSGi Editor (which is included) or a separate instance of IntelliJ IDEA or Eclipse.

Your Java code can get or set entity data or call additional domain methods with similar functionality in Java as in Gosu. For example, a **Message** object as viewed in the Java API has data getter and setter methods to get and set data. For properties, the objects have getter and setter methods. For example, the **Message** object has **getPayload** and **setPayload** methods that manipulate the **Payload** property. Additionally, the object has additional methods that trigger complex logic, such as the **reportAck** method.

Understanding and using the entity libraries is critical in the following situations.

- Java plugin development. Most ClaimCenter plugin implementations must access entity data or change entity data. Also, some plugin interface methods explicitly have entity data as method arguments or return values.
- Writing other Java classes that use entity data. Even if your Java code does not implement a plugin interface, your Java code can access entity data. You must write Gosu code that calls your new Java code.

In both cases, your code can perform the following actions.

- Take entity data as method arguments or return values
- Search for entity data
- Change entity data

When you are done writing your Java code, deploy your class files and JAR files.

Regenerate Java API libraries

To synchronize any changes that you make to the data model with your Java environment, you regenerate the Java API libraries.

About this task

Whenever you change the data model due to extensions or customizations, you must regenerate the entity libraries and compile your Java code against the latest libraries.

Procedure

1. In Windows, open a command prompt.
2. From the ClaimCenter installation directory, run the following command.

```
gwb genJavaApi
```

Results

The command generates Java entity interfaces and typelist classes in libraries in the following location.

```
ClaimCenter/java-api/lib
```

Note that the command does not regenerate the Java API Reference documentation.

Java API reference documentation

The Java API Reference documentation describes the base configuration types available from Java, such as entity and typelist types. It also includes definitions for Java plugin interfaces. The reference contents are static and cannot be regenerated to include your own data model changes.

View the Java API Reference at the following location.

```
ClaimCenter/javadoc/
```

Accessing entity instances from Java

In Gosu, you can refer to an entity type using the syntax `entity.ENTITYNAME` or merely the entity name because the package `entity` is always in scope.

In Java, the class name is the same as in Gosu but the package `entity` is not always in scope. Reference a type directly by its fully-qualified name, or use a Java `import` statement.

Entity properties appear from Java as getter and setter methods. Getter and setter methods are methods to get or set properties using method names that start with `get` or `set`. For example, a readable and writable property named `MyField` appears in Java as methods called `getMyField` and `setMyField`. Read-only properties do not expose a `set` method on the object. If the property named `MyField` contains a value of type `Boolean`, it appears in the interface as `isMyField` instead of `getMyField`.

```
address.setFirstName("John");
lastName = address.getLastName();
tested = someEntity.isFieldname();
```

Most entity methods on entity instances appear as regular methods in Java.

```
claim.addEvent("MyCustomEventName");
```

However, neither Gosu enhancement properties nor Gosu enhancement methods are available directly on the Java entity class. To access Gosu enhancements, you must use reflection APIs.

Accessing typecodes from Java

The Java API exposes typelists and typecodes as Java classes. To access a typecode, first get a reference to the appropriate typelist class in the `typekey` package using the same name as in Gosu. For example, `typekey.Address`. Reference a type directly by its fully-qualified name, or use a Java `import` statement.

To get a specific typecode instance from the typelist, use the static properties on a typelist that represent a typecode. The typecode static instances have the `TC_` prefix, just like from Gosu. You do not need to instantiate any typecode object or call any special methods to access the instance from Java.

In the rare cases where you need to get a typecode reference from a `String` value known only at run time, use the `getTypeKey` method on the typelist class.

```
tc = typekey.ExampleTypelist.getTypeKey(typeCodeString)
```

Because the `getTypeKey` method uses reflection to access data at run time, the Gosu editor cannot validate typecode code values at compile time. For this reason, it is best if possible to avoid using reflection for accessing `typekey` instances.

To compare typecode instances for equality, you can either use the `equals` method or use the `==` (double equals) operator.

Getting a reference to an existing bundle from Java

To use entity instances, in many cases you need a reference to a bundle. A bundle is a programmatic abstraction that represents one database transaction.

There are some programming contexts in ClaimCenter in which there is a current database transaction, which means there is a current bundle. For example, the following code contexts include a current bundle.

- Code called from business rules
- Plugin interface implementation code, or code called by a plugin implementation
- Most PCF user interface application code

There are many contexts in which it is unsafe to commit a bundle. For example, all plugin code and most PCF code.

To get the current bundle from Java, use the same API as in Gosu.

```
gw.pl.persistence.core.Bundle b = gw.transaction.Transaction.getCurrent();
```

If there is no current bundle, you must create a bundle before creating entity instances or updating entity instances that you get from a database query.

Creating a new bundle from Java

In general, Java code does not need to create a new bundle because there is already a bundle to use for that kind of code context. There are rare cases in which you need to create a new bundle, but only do this if necessary.

You can directly create a new bundle using the `newBundle` method on the `Transaction` class.

```
import gw.pl.persistence.core.Bundle;
import gw.transaction.Transaction;

...
Bundle bundle = Transaction.newBundle();
```

Additionally, you can use the `runWithNewBundle` method, which is the same as in Gosu for this task. However, the corresponding Gosu method takes a Gosu block as an argument. From Java, the syntax is slightly different, using an anonymous class instead of a Gosu block.

```
gw.transaction.Transaction.runWithNewBundle(new Transaction.BlockRunnable() {  
    @Override  
    public void run(Bundle bundle) {  
        // Your code here...  
  
        // The bundle commits automatically if no exceptions occur before the end of the run method  
    }  
});
```

Creating a new entity instance from Java

To instantiate an entity instance, simply call the constructor on the class, as you would in Gosu or typical Java code. However, a bundle argument is required, even if there is a current bundle. In contrast, the bundle argument is often optional in Gosu. To get the current bundle, call `Transaction.getCurrent()`.

From Java plugin code, there is a current bundle.

For example, in programming contexts in which there is a current bundle, create a new `Address` entity instance with the following code.

```
import gw.pl.persistence.core.Bundle;  
import entity.Address;  
import gw.transaction.Transaction;  
  
...  
Bundle b = Transaction.getCurrent();  
Address a = new Address(b);
```

Querying for entity data from Java

In the Java API, if you need to find entity instances, use the Query Builder APIs.

IMPORTANT: Using the Query Builder APIs to create a query is different in Java and Gosu. In Java, you must use the `gw.api.database.Quries.createQuery` method rather than `gw.api.database.Query.make`.

See also

- “Query builder APIs” on page 745

Using reflection to access Gosu classes from Java

From Java, you use reflection to access Gosu types. *Reflection* means asking the type system about types at run time to get data, set data, or invoke methods. Reflection is a powerful way to access type information at run time, but is not type safe. For example, language elements such as class names and method names are manipulated as `String` values. Be careful to correctly type the names of classes and methods because the Java compiler cannot validate the names at compile time.

To use reflection from Java, you use the same utility class `gw.lang.reflect.ReflectUtil` that provides reflection to Gosu classes. The `ReflectUtil` class contains methods to get information about a class, get information about members (properties and methods), and invoke static class methods or instance methods.

To access Gosu API classes from Java, including the classes that you use to implement reflection, you must add `gosu-core-api-release.jar` to your Java project's class path.

To run the Java code that accesses Gosu classes, you must be in the ClaimCenter application context. You cannot run that code from your Java IDE because the IDE project does not initialize Gosu. When ClaimCenter starts, it performs that initialization. You can test the Java class and its methods by running them in Guidewire Studio with a running server.

Improving type safety in using reflection

If you need to call methods on a Gosu class from Java, the following general steps increase the degree of type safety.

1. Create a Java interface containing only those methods that you need to call from Java. For getting and setting properties, define the interface using getter and setter methods, such as `getMyField` and `setMyField`.

```
package doc.example.reflection.java;

public interface MyJavaInterface {
    public String getMyField();
    public void setMyField(String value);
}
```

2. Compile the interface and place the class file in its complete hierarchy in the `ClaimCenter/modules/configuration/plugins/Gosu/classes` folder. If the interface extends a plugin interface, use the `ClaimCenter/modules/configuration/plugins/shared/classes` folder instead.
3. Create a Gosu class that implements that interface. The Gosu class can contain additional methods as well as the interface methods.

```
package doc.example.reflection.gosu

class MyGosuClass implements doc.example.reflection.java.MyJavaInterface {
    public static function myMethod(input : String) : String {
        return "MyGosuClass returns the value ${input} as the result!"
    }

    override property get MyField():String{
        print("Get MyField called")
        return "test"
    }
    override property set MyField(value:String){
        print("hello")
    }
}
```

4. In code that requires a reference to the object, use `ReflectUtil` to create an instance of the Gosu class. The following approaches are available.

- Create an instance by using the `constructGosuClassInstance` method in `ReflectUtil`.

```
package doc.example.reflection.java;

import gw.lang.reflect.ReflectUtil;

public class MyJavaClass {

    ...

    public void accessGosuClass(String gosuClassName) {
        Object myGosuObject = ReflectUtil.constructGosuClassInstance(gosuClassName);
    }
}
```

- Define a method on an object that creates an instance, and call that method with the `invokeMethod` or `invokeStaticMethod` method in `ReflectUtil`, as appropriate.

To see a list of available methods on the created instance, use `ReflectUtil.invokeMethod(myGosuObject, "@IntrinsicType").getClass().getMethods()`. Getting the value of a property requires the `@` symbol before the property name. Alternatively, use the `getProperty` method.

Note: In both cases, any new object instances at compile time that are returned by `ReflectUtil` have the type `Object`.

5. Downcast the new instance to your new Java interface and assign a new variable of that interface type.

```
MyJavaInterface myJavaInterface = (MyJavaInterface) myGosuObject;
```

You now have an instance of an object that conforms to your interface.

6. Call methods on the Gosu instance as you normally would from Gosu. The getters, setters, and other method names are defined in your interface, so the method calls are type safe. For example, the Java compiler can protect against misspelled method names.

```
myJavaInterface.setMyField("New value");
```

Calling a Gosu method from Java by reflection

If you need to call a method by reflection without using the previous technique, you can pass parameter values as additional arguments to the `invokeMethod` or `invokeStaticMethod` method in `ReflectUtil`.

The following example Gosu class defines a static method:

```
package doc.example.reflection.gosu

class MySimpleGosuClass {
    public static function myMethod(input : String) : String {
        return "MyGosuClass method myMethod returns the value ${input} as the result!"
    }
}
```

The following Java code calls the static method defined in `MyGosuClass` by calling the `ReflectUtil.invokeStaticMethod` method:

```
package doc.example.reflection.java;

import gw.lang.reflect.ReflectUtil;

public class MyNonTypeSafeJavaClass {

    public void callStaticGosuMethod() {
        // Call a static method on a Gosu class
        Object o = ReflectUtil.invokeStaticMethod("doc.example.reflection.gosu.MySimpleGosuClass", "myMethod", "hello");

        // Print the return result
        System.out.println((String) o);
    }
}
```

Using Gosu enhancement properties and methods from Java

Gosu enhancements are a way of adding properties and methods to a type, even if you do not control the source code to the class. Gosu enhancements are a feature of the Gosu type system. Gosu enhancements are not directly available on types from Java.

You can use the `gw.lang.reflect.ReflectUtil` class to call enhancement methods and access enhancement properties. The syntax is more complex than it would be from Gosu. Because it uses reflection, it is less typesafe. There is more chance of errors at run time that were not caught at compile time. For example, if you misspell a method name passed as a `String` value, the Java compiler cannot catch the misspelled method name at compile time.

Class loading and delegation for non-OSGi Java

ClaimCenter uses rules to determine how to load Java classes.

Java class loading rules

To load custom Java code into Gosu or to access Java classes from Java code, the Java virtual machine must locate the class file with a class loader. Class loaders use the fully qualified package name of the Java class to determine how to access the class.

ClaimCenter uses the rules in the following list to load Java classes, choosing the first rule that matches and not using the rules later in the list.

1. General delegation classes

The following classes delegate load, which means to delegate class loading to a parent class loader.

- `javax.*` - Java extension classes

- org.xml.sax.* - SAX 1 & 2 classes
- org.w3c.dom.* - DOM 1 & 2 classes
- org.apache.xerces.* - Xerces 1 & 2 classes
- org.apache.xalan.* - Xalan classes
- org.apache.commons.logging.* - Logging classes used by WebSphere

2. Internal classes

If the class name starts with `com.guidewire`, ClaimCenter delegate loads in general, but there are some internal classes that locally load.

Java code that you deploy must never access any internal classes other than supported classes and documented APIs. Using internal classes is dangerous and unsupported. If in doubt about whether a class is supported, immediately ask Customer Support. Never use classes in the `com.guidewire.*` packages.

3. All your classes

Any remaining non-internal classes load locally.

Java classes that you deploy must not have a fully qualified package name that starts with `com.guidewire`. Additionally, never rely on classes with that prefix because they are internal and unsupported.

Java class delegate loading

If the ClaimCenter class loader delegates Java class loading, ClaimCenter requests the parent class loader to load the class, which is the ClaimCenter application. If the ClaimCenter application cannot find the class, the ClaimCenter class loader attempts to load the class locally.

Deploying non-OSGi Java classes and JAR files

The following deployment instructions apply to non-OSGi Java class files or JAR files, regardless of whether your code uses Guidewire entity data. Carefully deploy Java class files and JAR files in the locations defined in this topic. Putting files in other locations is dangerous and unsupported.

Locations for Java classes and libraries

Place your non-OSGi Java classes and libraries in the following locations. Subdirectories must match the package hierarchy.

If the class implements a single ClaimCenter plugin interface, in Guidewire Studio, you can define a separate plugin directory in the Plugins registry for that interface. In the following paths, `PLUGIN_DIR` represents the plugin directory that you define in the Plugins registry. If the **Plugin Directory** field in the Studio Plugins registry is empty, the default is the plugin directory name `shared`.

The following special values exist for `PLUGIN_DIR`.

- If your code uses entities:
 - To share code among multiple plugin interfaces, or to share between plugin code and non-plugin code, use the value `shared`.
 - If you are not implementing a plugin interface, use the value `Gosu`. Notice the capitalization of `Gosu`. For example, if the only code that calls your Java code is your own Gosu code, use the value `Gosu`.
- If your code does not use entities:
 - To share code among multiple plugin interfaces, or to share between plugin code and non-plugin code, use the value `shared/basic`.
 - If you are not implementing a plugin interface, use the value `Gosu/basic`. Notice the capitalization of `Gosu`. For example, if the only code that calls your Java code is your own Gosu code, use the value `Gosu/basic`.

The ClaimCenter/modules/configuration/plugins/Gosu directory has subdirectories called gclasses, gresources, and idea-gclasses. These subdirectories are for internal use as compilation output paths. Never use, modify, or rely upon the contents of the gclasses, gresources, and idea-gclasses directories.

Place non-OSGi Java classes in the following locations as the root directory of directories organized by package.

```
ClaimCenter/modules/configuration/plugins/PLUGIN_DIR/classes
```

The following example references a class file using the fully qualified name mycompany.MyClass.

```
ClaimCenter/modules/configuration/plugins/PLUGIN_DIR/classes/mycompany/MyClass.class
```

Place your Java libraries and any third-party libraries in the following locations.

```
ClaimCenter/modules/configuration/plugins/PLUGIN_DIR/lib
```

Deploy an example Java class with no plugins and no entities

Test the use of Java in your ClaimCenter environment by deploying a simple Java class.

About this task

The following example demonstrates a simple use of Java that does not use entity data and does not implement a plugin interface.

Procedure

1. Launch IntelliJ IDEA with OSGi Editor by running the following command at a command prompt.

```
gwb pluginStudio
```

2. In IntelliJ, create a new empty Java project.
3. In IntelliJ, create a new class doc.example.java.SimpleClass.

```
package doc.example.java;  
public class SimpleClass {  
    public int add (int x, int y){  
        return (x + y);  
    }  
}
```

4. Compile the class or build the project.
5. Copy the compiled class file from the following path.

```
PROJECT/out/production/PROJECT_FOLDER/doc/example/java/SimpleClass.class
```

6. Place a copy in the deployment directory.

```
ClaimCenter/modules/configuration/plugins/Gosu/classes/doc/example/java/MyClass.class
```

7. Open the Gosu Scratchpad and type the following code.

```
// Instantiate your Java class  
var obj = new doc.example.java.SimpleClass()  
  
// Call the method  
var methodResult = obj.add(2,3)  
  
print (methodResult)
```

8. Run the Gosu Scratchpad code.

The example prints the following output.

Using IntelliJ IDEA with OSGi editor to deploy an OSGi plugin

To use IntelliJ IDEA with OSGi Editor to deploy an OSGi plugin, you must perform multiple tasks, starting with the following ones.

Generating Java API libraries before using OSGi

Before starting work with Java and OSGi, regenerate the ClaimCenter Java API libraries. Without Guidewire Studio or IntelliJ IDEA with OSGi editor open, open a command prompt in the application root directory and type the following command.

```
gwb genJavaApi
```

This requirement is independent of which Java IDE that you use.

Launching IntelliJ IDEA with OSGi editor

For OSGi plugin development, Guidewire recommends that you use the included application IntelliJ IDEA with OSGi Editor. To launch IntelliJ IDEA with OSGi Editor, open a command prompt in the application root directory and type the following command.

```
gwb pluginStudio
```

If you use other Guidewire applications, such as Guidewire ContactManager, each Guidewire application includes its own version of IntelliJ IDEA with OSGi Editor. Be sure to run this command from the correct application product directory.

Set up a project with an OSGi plugin module

An OSGi plugin module must be in a project. Follow this procedure to create a project or use an existing empty project.

About this task

After you run the `gwb pluginStudio` command for the first time for a Guidewire product, IntelliJ starts with an empty workspace and no current project. You must create a project to contain your OSGi plugin. If you have previously run IntelliJ IDEA with OSGi Editor and have created an empty project, you can use that project. If you have a project that contains files, you must create a new project.

Procedure

1. Open a command prompt in the application root directory.
2. Generate the Java API libraries by typing the following command.

```
gwb genJavaApi
```

3. Launch IntelliJ with OSGi Editor by typing the following command.

```
gwb pluginStudio
```

If you have previously run IntelliJ IDEA with OSGi Editor and have an empty project that you want to use, do this step.

4. To add or import an OSGi module to an existing empty project, perform the following steps:
 - a) In the **Project Structure** dialog, select **Empty Project** and set the **Project name** field.
 - b) Add a module in the **Project Structure** dialog by clicking the plus sign (+).
 - c) For a new module, choose **New Module**, and then select the module type **OSGi Plugin Module**. In the **Module name** field, type the module name. Go to “Step 6”.

- d) To select an existing module to import, choose **Import Module**.
- e) Click **Next** repeatedly and confirm the settings for the imported module.
- f) Click **Finish**.
- g) Click **OK**.

You do not do the remaining steps in this procedure.

If you do not have an empty project, you must create a new project by doing all the following steps.

5. To create a new project, perform the following steps:
 - a) To create the first project in IntelliJ IDEA with OSGi Editor, click **Create New Project**. If IntelliJ starts with an existing project, click **File > New Project**.
 - b) In the list of module types, click **Guidewire** and then, in the main panel of the dialog box, click **OSGi Plugin Module**.
 - c) Click **Next**.
 - d) In the **Project name** field, type the project name. In the **Project location** field, type the project location.
Do not yet click **Finish**.
6. Change the settings on the new module.
 - a) Open the **More Settings** pane, if it is not already open.
 - b) Set the name of the new module as appropriate.
By default, the **Bundle Symbolic Name** field matches the name of the module.
 - c) Optionally, change the symbolic name to a different value.
The bundle symbolic name defines the main part of the output JAR file name before the version number.
 - d) Optionally, change the version of the bundle in the **Bundle Version** field.
7. Click **Finish**.
If IntelliJ started with an existing project, you must choose whether to replace that project in the current window or open a new IntelliJ window.
8. If this is a new project, you must set the project JDK.
 - a) Click **File > Project Structure**.
 - b) In the **Project Settings > Project** section, set the **Project SDK** picker to your Java JDK.
 - c) If there is no Java JDK listed, click the **New** button. Click **JDK**, and then navigate to and select the Java JDK that you want to use. Next, set the **Project SDK** picker to your newly created Java JDK configuration.
 - d) Click **OK**.

What to do next

- “Create an OSGi-compliant class that implements a plugin interface” on page 57

Create an OSGi-compliant class that implements a plugin interface

Before you begin

- “Set up a project with an OSGi plugin module” on page 56

Procedure

1. If you have not previously generated the Java libraries, without Guidewire Studio or IntelliJ IDEA with OSGi editor open, type the following command at a command prompt in the ClaimCenter root directory.

```
gwb genJavaApi
```

2. If IntelliJ with OSGi Editor is not open, type the following command at a command prompt in the ClaimCenter root directory.

```
gwb pluginStudio
```

3. In IntelliJ IDEA with OSGi Editor, navigate under your OSGi module to the `src` directory.
4. Create new subpackages as necessary by right-clicking `src` and choosing **New > Package**. For example, create a `mycompany` package.
5. Create an OSGi-compliant class that implements a plugin interface.
 - a) Right-click the package for your class.
 - b) Choose **New > New OSGi plugin**.

IntelliJ IDEA with OSGi Editor opens a dialog.

 - c) In the **Plugin class name** field, enter the name of the Java class that you want to create. For example, type `DemoStartablePlugin`.
 - d) In the **Plugin interface** field, enter the fully qualified name of the plugin interface that you want to implement. Alternatively, to choose from a list, click the ellipsis (...) button. Type some of the name or scroll to the required plugin interface. Select the interface and then click **OK**. For example, type `IStartablePlugin`.

IntelliJ IDEA with OSGi Editor displays a dialog **Select Methods to Implement**. By default, all methods are selected.

 - e) Click **OK**.

IntelliJ IDEA with OSGi Editor displays your new class with stub versions of all required methods.
6. If the top of the Java class editor has a yellow banner that says **Project SDK is not defined**, you must set your project SDK to a JDK. If the SDK settings are uninitialized or incorrect, your project shows many compilation errors in your new Java class.
7. Add code to the stub methods.
For example, use the code in “Example startable plugin in Java using OSGi” on page 58.

Results

Your class is ready for compiling and installing.

What to do next

If you have several tightly related OSGi plugin implementations, you can optionally deploy them in the same OSGi bundle. To implement additional plugins in this project, repeat this procedure for each plugin implementation class. For example:

A messaging destination implements the `MessageTransport` interface. A single messaging destination optionally also implements the `MessageReply` plugin interface. If these related plugin implementations have shared code and third-party libraries, you could deploy them in the same OSGi bundle that encapsulates messaging code for a messaging destination.

- “Compile and install your OSGi plugin as an OSGi bundle” on page 60

Example startable plugin in Java using OSGi

The following example defines a class that implements the `IStartablePlugin` interface. The class prints a message to the console for each application call to each plugin method. Use the example to confirm that you can successfully deploy a basic OSGi plugin using IntelliJ IDEA with OSGi Editor.

Create a class with the fully qualified name `mycompany.DemoStartablePlugin` with the following contents.

```
package mycompany;
```

```
import gw.api.startable.IStartablePlugin;
import gw.api.startable.StartablePluginCallbackHandler;
import gw.api.startable.StartablePluginState;
import org.osgi.service.component.annotations.Activate;
import org.osgi.service.component.annotations.Component;
import org.osgi.service.component.annotations.ConfigurationPolicy;
import org.osgi.service.component.annotations.Deactivate;

import java.util.Map;

@Component(service = IStartablePlugin.class, configurationPolicy = ConfigurationPolicy.REQUIRE)
public class DemoStartablePlugin implements IStartablePlugin {

    private StartablePluginState _state = StartablePluginState.Stopped;

    private void printMessageToGuidewireConsole(String s) {
        System.out.println("*****");
        System.out.println("***** -> STARTABLE PLUGIN METHOD = " + s);
        System.out.println("*****");
    }

    @Activate
    public void activate(Map<String, Object> config) {
        printMessageToGuidewireConsole("activate -- OSGi plugin init");
        // The Map contains the plugin parameters defined in the Plugins registry in Studio
        // There are additional OSGi-specific parameters in this Map if you want them.
    }

    @Deactivate
    public void deactivate() {
        printMessageToGuidewireConsole("deactivate -- OSGi plugin shutdown");
    }

    // Other IStartablePlugin interface methods...

    @Override
    public void start(StartablePluginCallbackHandler startablePluginCallbackHandler, boolean b)
        throws Exception {
        printMessageToGuidewireConsole("start");
    }

    @Override
    public void stop(boolean b) {
        printMessageToGuidewireConsole("stop");
    }

    @Override
    public StartablePluginState getState() {
        printMessageToGuidewireConsole("getState");
        return _state;
    }
}
```

About the OSGi Ant build script

The main script for OSGi compilation and deployment is an Ant build script. You run this script from the **Ant** pane in IntelliJ with OSGi Editor.

The `install` target of the Ant build script performs the following operations.

1. Compiles Java code.
2. Generates OSGi metadata.
3. Packages code and library dependencies into an OSGi bundle.
4. Installs an OSGi bundle to the correct ClaimCenter bundles directory as defined in the OSGi plugin project's `build.properties` file. The script copies your final JAR file to the following ClaimCenter directory.

ClaimCenter/modules/configuration/deploy/bundles

Compile and install your OSGi plugin as an OSGi bundle

To use your OSGi plugin with ClaimCenter, you compile the plugin and then use Studio to register the plugin.

Before you begin

Create an OSGi-compliant Java class in a project in IntelliJ with OSGi Editor. Either perform the steps in “Create an OSGi-compliant class that implements a plugin interface” on page 57 or use a class from the Java examples in the `java-examples.zip` file, which is located in the `ClaimCenter` installation directory.

Procedure

1. If IntelliJ with OSGi Editor is not open, type the following command at a command prompt in the ClaimCenter root directory.

```
gwb pluginStudio
```

2. Open the OSGi Ant build script:

- a) Click **View > Tool Windows > Ant**.

The **Ant** pane opens in IntelliJ with OSGi Editor.

- b) If **OSGi plugin bundle** is not open in the Ant pane, click the plus sign (+) in the navigation bar on that pane. In **Select Ant Build File**, navigate to and select `build.xml` in the project folder. Then, click **OK**.

OSGi plugin bundle opens.

3. Run the Ant install command:

- a) Expand **OSGi plugin bundle**.

- b) Right-click **install** and click **Run Target**.

- c) To see the result of running the command, click **View > Tool Windows > Messages**. Expand the targets to see the messages from each step.

The command generates messages about compiling source files, generating files, copying files, building a JAR file, and finally copying that JAR file to the bundle deployment folder of ClaimCenter. If the command succeeds, the following output appears.

```
Ant build completed successfully
```

If the build completes with warnings, check the warnings and correct them if necessary.

4. Switch to, or open, Guidewire Studio, not the IntelliJ IDEA with OSGi Editor. Navigate in the **Project** pane to the path **configuration > deploy > bundles**. Confirm that you see the newly-deployed file `YOUR_JAR_NAME.jar`.

The JAR file name is based on the module symbolic name followed by the version. If the JAR file is not present at that location, check the Ant console output for the directory path where the script copied the JAR file. If you recently installed a new version of ClaimCenter at a different path, or moved your Guidewire product directory, you must immediately update your OSGi settings in your OSGi project.

5. In Guidewire Studio, register your OSGi plugin implementation.

Registering a plugin implementation defines where to find a plugin implementation class and what interface the class implements.

- a) In the **Project** window, navigate to **configuration > config > Plugins > registry**.

- b) Right-click the **registry**, and choose **New > Plugin**.

- c) In the plugin dialog, enter the plugin name in the **Name** field.

For plugin interfaces that support only one implementation, enter the interface name without the package. For example: `IBaseUrlBuilder`. The text that you enter becomes the basis for the file name that ends in `.gwp`. The `.gwp` file represents one item in the Plugins registry.

If the plugin interface supports multiple implementations, like messaging plugins or startable plugins, this name can be any name of your choice.

If you are registering a messaging plugin, the plugin name must match the plugin name in fields in the separate **Messaging** editor in Studio.

For this demonstration implementation of the `IStartablePlugin` interface, enter the plugin name `DemoStartablePlugin`.

- d) In the plugin dialog, next to the **Interface** field, click the ellipsis (...), find the interface class, and select it. If you want to type the name, enter the interface name without the package.

For this demonstration implementation of the `IStartablePlugin` interface, find or type the interface name `IStartablePlugin`.

- e) Click **OK**.

- f) In the Plugins registry main pane for the new plugin, click the plus sign (+) and click **Add OSGi Plugin**.

- g) In the **Service PID** field, type the fully qualified Java class name for your OSGi implementation class.

Note that this name is different from the bundle name or the bundle symbolic name. The ellipsis (...) button does not display a picker to find available OSGi classes, so you must type the class name in the field.

For this demonstration implementation of the `IStartablePlugin` interface, type the fully-qualified class name `mycompany.DemoStartablePlugin`.

- h) Set any other fields that your plugin implementation needs, such as environment, server, or plugin parameters.

- i) Make whatever other changes you need to make in Guidewire Studio. For example, if your plugin is a messaging plugin, configure a new messaging destination. You might need to make other changes such as changes to your rule sets or other related Gosu code.

- j) Start the server.

Click the **Run** or **Debug** menu items or buttons.

Alternatively, run the QuickStart server from the command prompt. Open a command prompt in the root application directory and run the following command.

```
gwb runServer
```

Example

“Example startable plugin in Java using OSGi” on page 58 shows an example startable plugin. If you used that example, closely read the console messages during server start-up. The example OSGi startable plugin prints messages during server start-up. These messages demonstrate that ClaimCenter successfully called your OSGi plugin implementation.

What to do next

- “Configuring third-party libraries in an OSGi plugin” on page 61

Configuring third-party libraries in an OSGi plugin

If your OSGi plugin code uses third-party libraries, there are two deployment options, depending on your type of third-party JAR file.

- Embed the JAR inside your OSGi bundle. You can embed any Java library in your OSGi plugin bundle. The library is not required to be an OSGi bundle, but OSGi third-party libraries are also supported. Deploy your library JAR in the module directory within the `inline-lib` subdirectory.
- Deploy as a separate OSGi bundle. This option requires a third-party JAR to support OSGi. If your library contains a properly-configured OSGi bundle with OSGi properties in the manifest, you can optionally deploy it as a separate OSGi bundle outside your main OSGi plugin bundle. For use within your OSGi plugin project, deploy your library JAR in the module directory within the `lib` (not `inline-lib`) subdirectory.

Deploy a third-party OSGi-compliant library as a separate bundle

About this task

If your library contains a properly configured OSGi bundle with OSGi properties in the manifest, you can optionally deploy the library as a separate OSGi bundle outside your main OSGi plugin bundle.

Procedure

1. Put any third-party OSGi JAR files in the `lib` (not `inline-lib`) folder inside your OSGi module in IntelliJ IDEA with OSGi Editor.
2. Write code that uses the third-party JAR.
3. Confirm that there are no compilation errors.
4. Open a command prompt at the root of your module and run the following command.

```
ant install
```

The tool generates various messages, and concludes with the following output.

```
BUILD SUCCESSFUL
```

Results

The `ant install` script copies your bundle JAR files to the `ClaimCenter/deploy/bundles` directory.

Embed a third-party Java library in your OSGi bundle

To use classes and methods in a third-party library in your OSGi bundle, you create a manifest file and configure the classes to import.

About this task

You can embed any Java library in your OSGi plugin bundle. The library is not required to be an OSGi bundle, but OSGi third-party libraries are also supported. Your bundle manifest might require special configuration for how to import the Java packages that your embedded libraries reference.

For example, if you want to use a third-party library that has 100 classes, you might use only 5 of the classes and only some of the methods on those classes. If the classes and methods that you use rely only on available and embedded libraries, there is no problem at run time.

Some of the third-party library classes that you do not use might have dependencies on external classes. The library might use a method that relies on classes that are unavailable at run time. OSGi tries to avoid risk of run-time errors by ensuring at server start-up that all required classes exist. Even if you never call those methods, there are errors on server start-up because OSGi verifies that all libraries are available, including ones that you never call.

You can modify the configuration file to avoid these types of errors. Your modification to the file specifies ignoring unavailable packages during OSGi library validation phase on server start-up. The following instructions include techniques to mitigate these problems.

Procedure

1. Put any third-party JAR files in the `inline-lib` folder inside your OSGi module in IntelliJ IDEA with OSGi Editor.
2. Write code that uses the third-party JAR.
3. Confirm there are no compilation errors.
4. Open a command prompt at the root of your module and run the following command.

```
ant dist
```

The tool generates various messages, and concludes with the following output.

```
BUILD SUCCESSFUL
```

5. Open the file `MODULE_ROOT/generated/META-INF/MANIFEST.MF`.
6. In that file, check that the import package (`Import-Package`) header includes no unexpected packages. All classes of the embedded libraries are copied inside your bundle such that all library classes become part of your bundle. This process might cause the `Bnd` tool to generate unexpected imports to appear in the list.
7. If you see unexpected imports, change the import package instruction in the `bnd.bnd` file, rather than modifying the `MANIFEST.MF` file. Never directly change the file `MANIFEST.MF`. This manifest file is automatically generated.

In the `bnd.bnd` file, set the `Import-Package` instruction to a comma-delimited list of Java packages to ignore by including each with the exclamation point (!) prefix. The exclamation point means “not,” or, in this context, it means “exclude.” At the end of the line, add an asterisk (*) character as the last item in the list to import all other packages.

To ignore only the packages `javax.inject` and `sun.misc`, use the following line.

```
Import-Package=!javax.inject,!sun.misc,*
```

8. Deploy and test your OSGi plugin.
9. Look for server start-up errors.

These types of errors look similar to the following lines.

```
ERROR Server.OSGi Could not start bundle messaging-OSGi-plugins from reference:  
file:/D:/GuidewireApp801/webapps/pc/bundles/messaging-OSGi-plugins-1.0.0.jar  
org.osgi.framework.BundleException: The bundle "messaging-OSGi-plugins_1.0.0 [12]" could not  
be resolved. Reason: Missing Constraint: Import-Package: javax.inject; version="0.0.0"
```

10. If there are errors, repeat this procedure and add more Java packages to the `Import-Package` line in `bnd.bnd`.

Test an example of embedding third-party libraries in an OSGi plugin

This procedure shows how to use classes and methods in a specific third-party library in your OSGi bundle by creating a manifest file and configuring the classes to import.

About this task

The following steps demonstrate how to add version 22 of the Java library called Guava to your OSGi plugin.

Procedure

1. Download `guava-22.0.jar` from the Guava project web site. Next, move the `guava-22.0.jar` file to the `inline-lib` folder in your OSGi project in IntelliJ IDEA with OSGi Editor.
2. Write some Java code that uses this library.

```
import com.google.common.escape.Escaper;  
...  
// Use the class com.google.common.escape.Escaper in guava-22.0.jar  
Escaper escaper = XmlEscapers.xmlAttributeEscaper();  
s = escaper.escape(s);
```

3. From a command prompt, run the following command, to generate OSGi metadata.

```
ant dist
```

Various messages appear. If the generation succeeded, the final output is shown below.

```
BUILD SUCCESSFUL
```

4. Open the file MODULE_ROOT/generated/META-INF/MANIFEST.MF and check that Import-Package header does not include any unexpected packages.

This manifest file might look like the following text.

```
Manifest-Version: 1.0
Service-Component: OSGI-INF/com.qa.MessageTransportOSGiImpl.xml
Private-Package: com.google.common.html,com.google.common.net,com.google.common.collect,
    com.google.common.primitives,com.google.common.base,com.google.common.escape,com.qa,
    com.google.common.base.internal,com.google.common.cache,com.google.common.eventbus,
    com.google.common.util.concurrent,com.google.common.hash,com.google.common.io,
    com.google.common.xml,com.google.common.reflect,com.google.common.math,
    com.google.common.annotations
Bundle-Version: 1.0.0
Tool: Bnd-1.50.0
Bundle-Name: messaging-OSGi-plugins
Bnd-LastModified: 1382994235749
Created-By: 1.8.0_45 (Oracle Corporation)
Bundle-ManifestVersion: 2
Bundle-SymbolicName: messaging-OSGi-plugins
Import-Package: gw.pl.messaging.entity,gw.plugin.messaging,javax.annotation,javax.inject,sun.misc
Bundle-RequiredExecutionEnvironment: JavaSE-1.8
```

5. Note the following line in the manifest file.

```
Import-Package: gw.pl.messaging.entity,gw.plugin.messaging,javax.annotation,javax.inject,sun.misc
```

By default, the packages javax.inject and sun.misc are unavailable at run time. The package javax.inject is a JavaEE package that is not exported by default. If you install the generated OSGi bundle in its current form and start the server, the server generates the following error.

```
ERROR Server.OSGi Could not start bundle messaging-OSGi-plugins from reference:
    file:/D:/GuidewireApp801/webapps/pc/bundles/messaging-OSGi-plugins-1.0.0.jar
org.osgi.framework.BundleException: The bundle "messaging-OSGi-plugins_1.0.0 [12]" could not
    be resolved. Reason: Missing Constraint: Import-Package: javax.inject; version="0.0.0"
```

In this case, both problematic packages are optional.

6. In the bnd.bnd file, set the Import-Package instruction to a comma-delimited list of Java packages to ignore by including each with the exclamation point (!) prefix. Add an asterisk (*) character as the last item in the list to indicate to import all other packages. For example, to ignore packages javax.inject and sun.misc, use the following line.

```
Import-Package=!javax.inject,!sun.misc,*
```

7. Run the ant install tool.

The scripts embed the guava JAR in your OSGi bundle.

8. Open the MANIFEST.MF file and notice two changes.

- An additional Ignore-Package instruction
- The Import-Package instruction does not include the problematic packages

The file contents can be similar to the following text.

```
Manifest-Version: 1.0
Service-Component: OSGI-INF/com.qa.MessageTransportOSGiImpl.xml
Private-Package: com.google.common.html,com.google.common.net,com.google.common.collect,
    com.google.common.primitives,com.google.common.base,com.google.common.escape,com.qa,
    com.google.common.base.internal,com.google.common.cache,com.google.common.eventbus,
    com.google.common.util.concurrent,com.google.common.hash,com.google.common.io,
    com.google.common.xml,com.google.common.reflect,com.google.common.math,
    com.google.common.annotations
Ignore-Package: javax.inject,sun.misc
Tool: Bnd-1.50.0
Bundle-Name: messaging-OSGi-plugins
Created-By: 1.8.0_45 (Oracle Corporation)
Bundle-RequiredExecutionEnvironment: JavaSE-1.8
Bundle-Version: 1.0.0
Bnd-LastModified: 1382995392983
Bundle-ManifestVersion: 2
Import-Package: gw.pl.messaging.entity,gw.plugin.messaging,javax.annotation
Bundle-SymbolicName: messaging-OSGi-plugins
```

Advanced OSGi dependency and settings configuration

The IntelliJ IDEA with OSGi Editor application configures dependencies and several additional files related to module configuration and Ant build script. These files are created automatically.

You can edit these files directly in the file system if necessary without harming OSGi module settings. The only file that IntelliJ IDEA with OSGi Editor modifies is `build.properties`, to edit the bundle symbolic name and version.

OSGi dependencies

The IntelliJ IDEA with OSGi Editor creates the following dependencies.

- `PROJECT/MODULE/inline-lib` directory – Directory for third-party Java libraries to embed in your OSGi bundle.
- `PROJECT/MODULE/lib` directory – Directory for third-party OSGi-compliant libraries to use but deploy into a separate OSGi bundle.
- `ClaimCenter/java-api/lib` – The ClaimCenter Java API files that the `regenJavaApi` tool creates.

If you require any JAR files, put them in the `lib` or `inline-lib` directories as discussed earlier. You do not need to modify any build scripts to add JAR files.

If you add dependencies in the IntelliJ project from other directories, change the associated `build.xml` (Ant build script) to include those directories.

OSGi build properties

The `build.properties` file at the root directory of the module contains the bundle symbolic name, version, and the path to Guidewire product. From within IntelliJ IDEA with OSGi Editor, you can edit these fields but optionally you can directly modify this file as needed. The following is an example of the file.

```
Bundle-SymbolicName=messaging-OSGi-plugins
Bundle-Version=1.0.0
-gw-dist-directory=C:/ClaimCenter/
```

The Ant build script (`build.xml`) uses these properties.

OSGi bundle metadata configuration file

The OSGi bundle metadata configuration file is called `bnd.bnd`. The contents of this file configure how scripts in the IntelliJ IDEA with OSGi Editor application generate OSGi bundle metadata, such as the `MANIFEST.MF` file and other descriptor files.

By default, the file contains the following text.

```
Import-Package=*
DynamicImport-Package=
Export-Package=
Service-Component=*
Bundle-DocURL=
Bundle-License=
Bundle-Vendor=
Bundle-RequiredExecutionEnvironment=JavaSE-1.8
-consumer-policy=${range;[==,+)}
-include=build.properties
```

For the format of this file, refer to the following online third-party documentation.

<http://bnd.bndtools.org/chapters/790-format.html>

However, for the `Bundle-SymbolicName` and `Bundle-Version` properties, you must set or change those settings in the `build.properties` file. Both the `bnd.bnd` and `build.xml` file use those properties from the `build.properties` file.

You typically edit this file if you need to export some Java packages from your bundle, or you need to customize the `Import-Package` header generation.

OSGi build configuration Ant script

The build configuration Ant script (`build.xml`) relies on properties from the files `build.properties` and `bnd.bnd`.

For advanced OSGi configuration, you can modify the `build.xml` build script.

Update your OSGi plugin project after product location changes

Before you begin

After you initially configure a project within IntelliJ IDEA with OSGi Editor, the project includes information that references the Guidewire product directory on your local disk. If you move your product directory, you must update your project with the new product location on your local disk. Similarly, if you upgrade your Guidewire product to a new version with a different install path, you must update the OSGi project.

Procedure

1. If you need to move your OSGi project on disk, quit IntelliJ IDEA with OSGi Editor and move the files on disk.
2. Launch IntelliJ IDEA with OSGi Editor from your new Guidewire product location.
3. In IntelliJ IDEA with OSGi Editor, open your OSGi plugin project.
4. In IntelliJ IDEA with OSGi Editor, update the module dependency for your OSGi module.
 - a) Open the **Project Structure** window.
 - b) Click the **Modules** item in the left navigation.
 - c) In the list of modules to the right, under the name of your module, click **OSGi Bundle Facet**.
 - d) To the right of the **Guidewire product directory** text field, click the **Change** button.
 - e) Set the new disk path and click **OK**.
 - f) Click **OK** in the dialog. The tool updates IDE library dependencies and the `build.properties` file.
5. Test your new configuration.

part 2

Web services

Web services provide a language-neutral, platform-neutral mechanism for invoking actions or requesting data between applications across a network.

Web services define request-and-response APIs that enable one web-based application to call an API on another web-based application by using an abstracted, well-defined interface. A data format, the Web Service Description Language (WSDL), describes available web services that other systems can call by using the SOAP protocol. Many languages and third-party packages provide bindings implementations of WSDL and SOAP, including Gosu (built into ClaimCenter), Java, Perl, and other languages.

Remote systems call ClaimCenter web services by using the SOAP protocol (Simple Object Access Protocol). The SOAP protocol defines request/response mechanisms for translating a function call and its response into XML-based messages, typically across the standard HTTP protocol.

ClaimCenter conforms to the requirements defined by the WS-I Basic Security Profile 1.1. Consumers of ClaimCenter web services must also be compliant with the profile.

ClaimCenter web services are written in the Gosu programming language. ClaimCenter publishes foundational web services that, when configured with additional code, can be deployed in a production environment.

Publishing or consuming web services from Gosu

Gosu natively supports web services in two ways:

Calling web service APIs published by external applications

Write Gosu code that imports and calls web service APIs published by external systems. Gosu parses the web service definition (WSDL) for the service. Gosu uses the WSDL to create Gosu types that enable you to call the remote API. You can call methods on the API and access types from the WSDL, all by using Gosu syntax.

Publishing your Gosu code as new web service APIs

Write Gosu code that external systems call as a web service by using the SOAP protocol. Add a single line of code before the definition of the implementing Gosu class. The application parses the class methods, generates WSDL, and publishes the web service to external systems.

IMPORTANT: Your web service definition in WSDL defines a strict programmatic interface to external systems that use your service. The WSDL encodes the structure of all parameters and return values. After moving code into production, do not change the WSDL. For example, do not modify data transfer objects (DTOs) after going into production or after widely distributing the WSDL in a user acceptance testing (UAT) environment.

What happens during a web service call

For all types of web services, ClaimCenter converts the server's local Gosu objects to and from the flattened, text-based format that the SOAP protocol requires. These processes are called *serialization* and *deserialization*.

- Suppose that you write a web service in Gosu, publish it from ClaimCenter, and call it from a remote system. Gosu must deserialize the text-based request into a local object that your Gosu code can access. If one of your web service methods returns a result, ClaimCenter serializes that local, in-memory Gosu result object into a text-based reply to the remote system.
- Suppose that you use Gosu to call a web service hosted by an external system. Before calling the API, Gosu serializes any API parameters to convert a local object into a flattened form to send to the API. If the remote API returns a result, ClaimCenter deserializes the response into local Gosu objects for your code to examine.

Writing your own custom web service for each integration point is the best approach for maximum performance and maintainability. Guidewire strongly encourages you to write as many web services as necessary to provide APIs for each integration point.

For example, write new web services to communicate with a check printing service, a legacy financials system, reporting service, or document management system. External systems can query ClaimCenter to calculate values, to trigger actions, or to change data in the ClaimCenter database.

Publishing a web service can be as simple as adding a line of code called an *annotation* immediately before your Gosu class.

For consuming a web service, Gosu creates local objects in memory that represent the remote API. Gosu creates types for every object in the WSDL, and you can create these objects or manipulate properties on them.

Reference of all web services in the base configuration

The following tables list all the web services in the base configuration.

Application web services

Web Service Name	Description	More information
BulkInvoiceAPI	Allows external systems to submit bulk invoices.	"Bulk invoice integration" on page 445
ClaimAPI	Manipulates claim and exposure data. For example, "Claim-related web add documents to a claim, create a new claim or FNOL (first notice of loss), re-open an exposure, and add activities to claims."	"Claim services" on page 149
ClaimFinancialsAPI	Performs various actions on claim financials, such as loading reserves, payments, and recoveries from external systems, and updating the status of payments.	"Claim financials web service (ClaimFinancialsAPI)" on page 421
ContactAPI	Creates, updates, and deletes contacts. Also synchronizes ClaimCenter contact IDs with contact IDs in an external address book. If you configure ContactManager to connect to ClaimCenter, ContactManager uses this web service automatically.	"Contact web service APIs" on page 540
PCClaimSearchIntegrationAPI	Intended for use only by the PolicyCenter integration. Searches for claims related to a policy.	"Claim search web service for policy system integration" on page 478

Web Service Name	Description	More information
ServiceRequestAPI	Updates and gets information about service requests from vendor systems.	"Service request web service (ServiceRequestAPI) overview" on page 159

General web services

Web Service Name	Description	More information
AdminDataAPI	For Guidewire use only. Do not use.	
DataChangeAPI	Tool for rare cases of mission-critical data updates on running production systems.	<i>Administration Guide</i>
ImportTools	Imports administrative data from an XML file. You must use this only with administrative database tables (entities such as User). This system does not perform complete data validation tests on any other type of imported data.	"Importing administrative data" on page 123
LoginAPI	The WS-I web service LoginAPI is not used for authentication in typical use. It is only used to test authentication or force a server session for logging.	"Login authentication confirmation" on page 88
MaintenanceToolsAPI	Starts and manages various background processes. The methods of this web service are available only when the server run level is maintenance or higher.	"Maintenance tools web service" on page 124
MessagingToolsAPI	Manages the messaging system remotely for message acknowledgments error recovery. The methods of this web service are available only when the server run level is multiuser.	"Messaging tools web service" on page 404
PersonalDataDestructionAPI	Enables an external application to request the following: <ul style="list-style-type: none"> • Destruction of a contact's data by AddressBookUID or by PublicID • Destruction of a user by user name 	"Data destruction web service" on page 132
ProfilerAPI	Sends information to the built-in system profiler.	"The Profiler API web service" on page 137
ServerStateAPI	Provides information about the state of the server that runs the ClaimCenter application, such as the run level and the server name.	"Server state web service" on page 139
SystemToolsAPI	Performs various actions related to server run levels, schema and database consistency, module consistency, and server and schema versions. The methods of this web service are available regardless of the server run level.	"System tools web service" on page 140
TableImportAPI	Loads operational data from staging tables into operational tables. Typically you use this web service for large-scale data conversions, such as migrating claims from a legacy system to ClaimCenter prior to bringing ClaimCenter into production. Important methods of this web service	"Table import web service" on page 142

Web Service Name	Description	More information
	are available only when the server run level is maintenance.	
TemplateToolsAPI	Lists and validates Gosu templates available on the server.	“Template web service” on page 144
TypeListToolsAPI	Retrieves aliases for ClaimCenter typecodes in external systems.	“Mapping typecodes and external system codes” on page 129
WorkflowAPI	Performs various actions on a workflow, such as suspending and resuming workflows and invoking workflow triggers.	“Workflow web service” on page 145
ZoneImportAPI	Imports geographic zone data from a comma separated value (CSV) file into a staging table, in preparation for loading zone data into the operational table.	“Zone data import web service” on page 146

Publishing web services

You can write web service APIs in Gosu and access them from remote systems by using SOAP, the standard web services protocol. The SOAP protocol defines request and response mechanisms for translating a function call and its response into XML-based messages typically sent across computer networks over the standard HTTP protocol. Web services provide a language-neutral and platform-neutral mechanism for invoking actions on or requesting data from another application across a network. In the base configuration, ClaimCenter publishes web service APIs for use by external programs and Guidewire applications.

Web service publishing quick reference

The following list summarizes important features of web services published by Guidewire applications.

Basic publishing of a web service

Add the annotation `@WsiWebService` to the implementation class, which must be a Gosu class that does not inherit from any other class. This annotation supports one optional argument for the web service namespace. If you do not declare the namespace, Gosu uses the default namespace `http://example.com`.

See “Declaring the namespace for a web service” on page 76.

Does ClaimCenter automatically generate WSDL files from a running server?

Yes.

See “Generating and publishing WSDL” on page 80.

Does ClaimCenter automatically generate WSDL files locally if you regenerate the SOAP API files?

Yes.

See “Generating and publishing WSDL” on page 80.

Does ClaimCenter automatically generate JAR files for Java SOAP client code if you regenerate the SOAP API files?

No. Use the Java built-in utility `wsimport`. The documentation includes examples.

See “Calling a ClaimCenter web service from Java” on page 82.

Can ClaimCenter serialize Gosu class instances, sometimes called POGOs: Plain Old Gosu Objects?

Yes, but with certain requirements.

See “Data transfer objects” on page 75.

Can web services serialize or deserialize Guidewire entity instances?

No. To transfer entity data, create your own data transfer objects (DTOs) that contain only the data you need for the service. DTO objects can be either Gosu class instances or XML objects from XSD types.

See “Data transfer objects” on page 75.

Can ClaimCenter serialize an XSD-based type?

Yes. Add the XSD to the source tree, run the code generation command in Studio, and write Gosu using the `XmlElement` APIs.

See “Using predefined XSD and WSDL” on page 92.

Can you use GX models to generate XML types that can be arguments or return types?

Yes

Can web services serialize a GX model as an argument or return type?

Yes. The web services framework will generate matching XSD types in the WSDL.

Can web services serialize a Java object as an argument type or return type?

In most cases, no. Exceptions include Simple Java objects. Such objects include but are not limited to `Date`, `Double`, and `String`.

Can web services serialize an enumeration—typelist, Java enum, or Gosu enum—as an argument type or return type?

Yes.

See “"Exposing Typelists and Enumerations as String Values"”.

Can web services automatically throw `SOAPException` exceptions?

No. Declare the actual exceptions you want thrown. In general, this requirement reduces typical WSDL size. To declare the exceptions a method can throw, use the `@Throws` annotation.

```
class MyClass{
    @Throws(SOAPServerException,
        "If communication error or any other SOAP problem occurs.")
    public function myMethod() { ... }
}
```

Logging in to the server

Each web service request embeds necessary authentication and security information in each request. If you use Guidewire authentication, you must add the appropriate SOAP headers before making the connection.

Package name of web services from the SOAP API client perspective.

The package in which you put your web service implementation class defines the package for the web service from the SOAP API client perspective. The biggest benefit from this change is reducing the chance of namespace collisions in general. In addition, the web service namespace URL helps prevent namespace collisions.

See “Declaring the namespace for a web service” on page 76.

Calling a local version of the web service from Gosu

Gosu creates types that use the original package name to avoid namespace collisions.

API references in the package `wsi.local.ORIGINAL_PACKAGE_NAME`

In production environments, if the operation is exposed only as a web service, instantiate the web service implementation class directly and call the method. This technique avoids having multiple transactions process the same action, which can result in possible race conditions, CDCEs (concurrent data change exceptions), or rollback issues.

For testing purposes only, the local files must be rebuilt by running the following command from the ClaimCenter installation directory.

```
gwb genWsiLocal
```

Creating SOAP 1.2 custom fault subcodes

Custom fault subcodes are not supported in ClaimCenter. However, configuration code can throw a `gw.xml.ws.WsdlFault` exception, which includes properties that can be populated and subsequently retrieved.

```
setActorRole(String actorRole) // SOAP 1.1 actor role  
getActorRole() : String  
  
setCode(FaultCode code)  
getCode() : FaultCode  
  
setCodeQName(QName codeQName)  
getCodeQName() : QName  
  
setDetail(XmlElement detail)  
getDetail() : XmlElement
```

The properties returned by the `WsdlFault` will not be documented in the generated WSDL.

Web service publishing annotation reference

The following list describes annotations that relate to web service declaration and configuration.

@WsiAdditionalSchemas

Exposes additional schemas to web service clients in the WSDL. Use this annotation to provide references to schemas that might be required but are not automatically included.

See “Referencing additional schemas in your published WSDL” on page 95.

@WsiAvailability

Sets the minimum server run level for this service.

See “Specifying the minimum run level for a web service” on page 77.

@WsiCheckDuplicateExternalTransaction

Detects duplicate operations from external systems that change data.

See “Checking for duplicate external transaction IDs” on page 104.

@WsiExportable

Add this annotation on a Gosu class to indicate support for serialization with web services. The annotation accepts an optional `String` argument that specifies a namespace.

See “Data transfer objects” on page 75.

@WsiExposeEnumAsString

Instead of exposing typelist types and enumerations as enumerations in the WSDL, this annotation exposes them as `String` values.

See “Exposing typelists and enums as enumeration values and string values” on page 105.

@WsiInvocationHandler

Performs advanced implementation of a web service that conforms to externally-defined standard WSDL. This annotation is for unusual situations only. This approach limits protection against some types of common errors.

See “Using predefined XSD and WSDL” on page 92.

@WsiParseOptions

Adds validation of incoming requests using additional schemas in addition to the automatically generated schemas.

See “Validating requests using additional schemas as parse options” on page 96.

@WsiPermissions

Sets the required permissions for this service.

See “Specifying required permissions for a web service” on page 77.

@WsiReduceDBConnections

A database query triggers the retrieval of a database connection from the connection pool. The management of database connections is handled by the pool

The `@WsiReduceDBConnections` annotation configures the web service operations to retrieve and re-use a single database connection from the pool. Without the annotation, a connection is retrieved from and returned to the pool for each query.

In most scenarios, the `@WsiReduceDBConnections` annotation is not needed. Using this annotation can cause connections to become locked, reducing the efficiency of the connection pool and slowing web service execution.

Use this annotation in special situations only. An example is when a single web service performs a very large number of database queries, such that retrieving and returning a pool connection for each query would cause excessive thrashing.

@WsiRequestTransform

@WsiResponseTransform

Adds transformations of incoming or outgoing data as a byte stream. Use this annotation to add advanced layers of authentication or encryption. Contrast to the `@WsiRequestXmlTransform` and `@WsiResponseXmlTransform` annotations, which operate on XML elements.

See “Transformations on data streams” on page 100.

@WsiRequestXmlTransform

@WsiResponseXmlTransform

Adds transformations of incoming or outgoing data as XML elements. Use this annotation for transformations that do not require access to byte data. Contrast to the `@WsiRequestTransform` and `@WsiResponseTransform` annotations, which operate on a byte stream.

See “Request or response XML structural transformations” on page 89.

@WsiSchemaTransform

Transforms the generated schema that ClaimCenter publishes.

See “Transforming a generated schema” on page 89.

@WsiSerializationOptions

Adds serialization options for web service responses, for example supporting encodings other than UTF-8.

See “Setting response serialization options, including encodings” on page 99.

@WsiWebMethod

The `@WsiWebMethod` annotation can perform two actions.

- Overriding the web service operation name to something other than the default value, which is the method name.
- Suppressing a method from generating a web service operation even though the method has `public` access.

See “Overriding a web service method name or visibility” on page 77.

@WsiWebService

Declares a class as implementing a web service.

See “Publishing and configuring a web service” on page 76.

Data transfer objects

A web service can accept a Gosu object as an argument. It can also return a Gosu object. Such an object passed by a web service between remote processes is often referred to as a *data transfer object* or DTO.

Most integration points need to transfer only a subset of an object's properties and object graph. Do not pass large object graphs. Be aware of any objects that in edge cases might be very large in your deployed production system. If you try to pass too much data, production systems can potentially experience memory problems. Design web services to pass DTOs that contain only the properties needed by the integration point. For example, an integration point for a record's main contact name and phone number requires a DTO containing only those properties and the standard public ID property.

Web service arguments and return values can be of the following types:

Gosu class

An example is a Gosu class that contains properties for a particular integration point.

For a Gosu class to be used as a DTO, it must meet certain requirements. For example, the class must be declared as `final` and it must specify the `@WsiExportable` annotation.

```
package doc.example.ws.myobjects
uses gw.xml.ws.annotation.WsiExportable

@WsiExportable
final class MyCheckPrintInfo {
    var checkID : String
    var checkRecipientDisplayName : String
}
```

XSD types

An example is an XML object based on an XSD type.

Store an XSD file in the Gosu source code tree and then reference the XSD types by using the Gosu XML API. Optionally, define a Guidewire GX model that contains the desired subset of properties from one or more entity types. Then use the GX model to import or export XML data, as needed.

ClaimCenter generates a WSDL from the Gosu web service definitions that you create and their related DTO types by running the following command.

```
gwb genWsiLocal
```

In addition, ClaimCenter includes a variety of DTOs used by web services to integrate between individual Guidewire applications. These DTOs define the contract between the applications and can be modified if you need to expand the set of properties exchanged in an integration.

Requirements for a Gosu object used as a DTO

A web service can accept arguments and define return values that contain or reference instances of Gosu classes, sometimes called Plain Old Gosu Objects (POGOs).

However, the Gosu class must have the following qualities.

- The class must be declared as `final`.
- The class must specify the `@gw.xml.ws.annotation.WsiExportable` annotation. The annotation accepts an optional `String` argument to specify a namespace.
- The class cannot define a property that has a getter or setter method of an XSD type.
- The class cannot define a method with an argument or return value of an XSD type.

Committing entity data to the database using a bundle

Guidewire applications use bundles to track changes to entity data.

No default bundle exists for a web service that changes and persists entity data. If your web service modifies entity data, you must create your own bundle. To create a bundle, use the `runWithNewBundle` API.

A web service does not need to create a bundle if it only retrieves entity data and does not modify and persist that data.

Publishing and configuring a web service

To publish a web service, use the `@WsiWebService` annotation on a class. Gosu publishes the class automatically when the server runs. An example web service is implemented below.

```
package example
uses gw.xml.ws.annotation.WsiWebService

@WsiWebService
class HelloWorldAPI {
    public function helloWorld() : String {
        return "Hello!"
    }
}
```

Choose your package declaration carefully. The package in which you put your web service implementation class defines the package for the web service from the SOAP API client perspective.

Arguments and return values must be WS-I exportable. This includes simple types like `String`, as well as XSD types and Gosu classes declared `final` and that have a special annotation. Arguments of array types are not supported.

Declaring the namespace for a web service

Each web service is defined within a namespace, which is similar to a Gosu class or Java class within a package. Specify the namespace as a URL in each web service declaration. The namespace is a `String` that looks like a standard HTTP URL but represents a namespace for all objects in the service's WSDL definition. The namespace is not typically a URL for an actual downloadable resource. Instead it is an abstract identifier that disambiguates objects in one service from objects in another service.

A typical namespace specifies your company and perhaps other meaningful disambiguating or grouping information about the purpose of the service, possibly including a version number:

- `"http://mycompany.com"`
- `"http://mycompany.com/xsds/messaging/v1"`

You can specify the namespace for each web service by passing a namespace as a `String` argument to the `@WsiWebService` annotation.

```
package doc.example
uses gw.xml.ws.annotation.WsiWebService

@WsiWebService("http://mycompany.com")
class HelloWorldAPI {
    public function helloWorld() : String {
        return "Hello!"
    }
}
```

You can omit the namespace declaration entirely and provide no arguments to the annotation. If you omit the namespace declaration in the constructor, the default is `"example.com"`.

Most tools that create stub classes for web services use the namespace to generate a package namespace for related types.

Specifying the minimum run level for a web service

To set the minimum run level for the service, add the annotation `@WsiAvailability` and pass the run level at which this web service is first usable. The choices include DAEMONS, MAINTENANCE, MULTIUSER, and STARTING. The default value is NODAEMONS.

```
package doc.example

uses gw.xml.ws.annotation.WsiWebService
uses gw.xml.ws.annotation

@WsiAvailability(MAINTENANCE)
@WsiWebService
class HelloWorldAPI {
    public function helloWorld() : String {
        return "Hello!"
    }
}
```

Specifying required permissions for a web service

To add required permissions, add the annotation `@WsiPermissions` and pass an array of permission types (`SystemPermissionType[]`). The default permission is SOAPADMIN. If you provide an explicit list of permissions, you can choose to include SOAPADMIN explicitly or omit it. If you omit SOAPADMIN from the list of permissions, your web service must not require SOAPADMIN permission. If you pass an empty list of permissions, your web service must not require authentication at all. Web services for which authentication is not needed include ping-type operations.

The only supported permission types are permissions that are role-based (static), rather than data-based (requires an object). Use the *Security Dictionary* to make this determination. If the permission is role-based, you see the term *(static)* after the permission name.

The following example uses no authentication for a simple service you might use for debugging.

```
package doc.example

uses gw.xml.ws.annotation.WsiWebService
uses gw.xml.ws.annotation.WsiPermissions

@WsiPermissions({}) // A blank list means no authentication needed.
@WsiWebService
class HelloWorldAPI {
    public function helloWorld() : String {
        return "Hello!"
    }
}
```

Overriding a web service method name or visibility

Web service method names must be unique. Duplicate method names, even with differing signatures, are not supported.

You can override the names and visibility of individual web service methods in several ways.

Overriding the name of a web service method

By default, the web service operation name for a method is the name of the method. You can override the name by adding the `@WsiWebMethod` annotation immediately before the declaration for the method on the implementation class. Pass a `String` value for the new name for the operation.

```
@WsiWebMethod("newMethodName")
public function helloWorld() : String {
    return "Hello!"
}
```

Hiding a public web service method

Gosu publishes one web service operation for each method marked as `public`. Public methods can be hidden by specifying the `@WsiWebMethod` annotation and passing the value `true`. By default, public web service methods are visible. The `@WsiWebMethod` annotation can hide a public method, but it cannot make a non-public method visible.

```
@WsiWebMethod(true)
public function helloWorld() : String {
    return "Hello!"
}
```

Overriding the method name and visibility with a single `@WsiWebMethod` annotation

You can combine both of these features of the `@WsiWebMethod` annotation by passing both arguments.

```
@WsiWebMethod("newMethodName", true)
public function helloWorld() : String {
    return "Hello!"
}
```

Web service invocation context

In some cases your web service may need additional context for incoming or outgoing information. For example, to get or set HTTP or SOAP headers. You can access this information in your implementation class by adding an additional method argument to your class of type `WsiInvocationContext`. Make this new object the last argument in the argument list.

Unlike typical method arguments on your implementation class, this method parameter does not become part of the operation definition in the WSDL. Your web service code can use the `WsiInvocationContext` object to get or set important information as needed.

The following table lists the request properties on `WsiInvocationContext` objects and whether each property is writable. All of these properties are readable.

Property	Description	Writable
<code>HttpServletRequest</code>	A servlet request of type <code>HttpServletRequest</code> .	No
<code>MtomEnabled</code>	<p>A boolean value that specifies whether to support sending MTOM attachments to the web service client in any data returned from an API. The W3C Message Transmission Optimization Mechanism (MTOM) is a method of efficiently sending binary data to and from web services as attachments outside the normal response body.</p> <p>If <code>MtomEnabled</code> is <code>true</code>, ClaimCenter can send MTOM in results. Otherwise, MTOM is disabled. The default is <code>false</code>.</p> <p>This property does not affect MTOM data sent to a published web service. Incoming MTOM data is always supported.</p> <p>This property does not affect MTOM support where Gosu is the client to the web service request. See “MTOM attachments with Gosu as web service client” on page 122.</p>	Yes
<code>RequestHttpHeaders</code>	Request headers of type <code>HttpHeaders</code> .	No
<code>RequestEnvelope</code>	Request envelope of type <code>XmlElement</code> .	No
<code>RequestSoapHeaders</code>	Request SOAP headers of type <code>XmlElement</code> .	No

The following table lists the response properties on `WsiInvocationContext` objects and whether each property is writable. All of these properties are readable.

Property	Description	Writable
ResponseHttpHeaders	Response HTTP headers of type HttpHeaders.	The property value is read-only, but you can modify the object it references.
ResponseSoapHeaders	Response SOAP headers of type List<XmlElement>.	The property value is read-only, but you can modify the object it references.

The following table lists the output serialization property on WsiInvocationContext objects and whether the property is readable. The property is readable.

Property	Description	Writable
XmlSerializationOptions	XML serialization options of type XmlSerializationOptions.	Yes

You can use these properties to modify response headers from your web service and read request headers as needed.

For example, suppose you want to copy a specific request SOAP header XML object and copy to the response SOAP header object. Create a private method to get a request SOAP header XML object.

```
private function getHeader(name : QName, context : WsiInvocationContext) : XmlElement {
    for (h in context.RequestSoapHeaders.Children) {
        if (h.QName.equals(name)) {
            return h
        }
    }
    return null
}
```

From a web service operation method, declare the invocation context optional argument to your implementation class. You can then use code with the invocation context to get the incoming request header and add it to the response headers. The following code demonstrates this approach.

```
function doWork(..., context : WsiInvocationContext) {
    var rtnHeader = getHeader(TURNAROUND_HEADER_QNAME, context)
    context.ResponseSoapHeaders.add(rtnHeader)

    // Do the main work of your web service operation
}
```

Web service class life cycle and variables scoped locally to a request

ClaimCenter does not instantiate a web service implementation class on server startup. ClaimCenter instantiates a web service implementation class on the first request from an external web service client for that specific service. That one object instance is shared across all requests and sessions on that server for the lifetime of the ClaimCenter application.

Because multiple threads can share this object, careful use of static and single-instance variables is required. To ensure thread safety, use concurrency APIs at all times.

For web service request-based data storage, use the concurrency class gw.xml.ws.WsiRequestLocal. If you use this class to create a web service implementation class instance variable, only a single instance of the variable will exist. The variable's get and set methods manage access to the variable in a thread-safe way.

If a web service encounters a concurrency issue, it returns a ConcurrentDataChangeException or SoapRetryableException. The issue can usually be resolved by retrying the web service request.

Note: For Gosu coding that is not in a web service implementation class, you can use the standard Gosu concurrency classes RequestVar and SessionVar in the gw.api.web package. However, these classes do not work in web service implementation classes.

Define a web request local scoped variable

Procedure

1. Decide what type of object you want to store in your request local variable. In your type declaration for the variable, parameterize the `WsiRequestLocal` class with the type you want to store. For example, if you plan to store a `String` object, declare your variable to have the type `WsiRequestLocal<String>`.
2. In your web service implementation class, define a private variable that uses the parameterized type.

```
private var _reqLocal = new WsiRequestLocal<String> ( )
```

3. In your implementation class, use the `get` and `set` methods to get or set the variable.

```
var loc = _reqLocal.get()  
...  
_reqLocal.set("the new value")
```

If you call `get` before calling `set` in the same web service request, the `get` method returns the value `null`.

Generating and publishing WSDL

Your web service definition in the WSDL defines a strict programmatic interface to external systems that use your web service.

The WSDL encodes the structure of all parameters and return values. After moving code into production, be careful not to change the WSDL. For example, do not modify data transfer objects (DTOs) after going into production or widely distributing the WSDL in a user acceptance testing (UAT) environment.

Getting WSDL from a running server

Typically, you get the WSDL for a web service from a running server over the network. This approach encourages all callers of the web services to use the most current WSDL. In contrast, generating the WSDL and copying the files risks callers of the web service using outdated web-service definitions. You can get the most current WSDL for a web service from any computer on your network that publishes the web service. In a production system, code that calls a web service typically resides on computers that are separate from the computer that publishes the web service.

ClaimCenter publishes a web service WSDL at the following URL.

```
SERVER_URL/WEB_APP_NAME/ws/WEB_SERVICE_PACKAGE/WEB_SERVICE_CLASS_NAME?WSDL
```

A published WSDL can make references to schemas at the following URL.

```
SERVER_URL/WEB_APP_NAME/ws/SCHEMA_PACKAGE/SCHEMA_FILENAME
```

For example, ClaimCenter generates and publishes the WSDL for web service class `gw.webservice.cc.cc1000.contact.AddressAPI` at the following location on a local server with web application name `cc`.

```
http://localhost:8080/cc/ws/gw/webservice/cc/cc1000/contact/AddressAPI?WSDL
```

WSDL and schema browser

If you publish web services from a Guidewire application, you can view the WSDL and a schema browser available at the following URL.

```
SERVER_URL/WEB_APP_NAME/ws
```

For example:

```
http://localhost:8080/cc/ws
```

The `WsbsevicesHideListPage` configuration parameter controls whether this information is available. If this parameter is set to `true`, this URL displays a blank page.

If this parameter is set to `false`, you can browse the web page for:

- Document/Literal Web Services
- Supporting Schemas
- Generated Schemas
- Supporting WSDL files

Example WSDL

The following example demonstrates how a simple Gosu web service translates to WSDL. This example uses the following simple example web service.

```
package example
uses gw.xml.ws.annotation.WsiWebService
@WsiWebService
class HelloWorldAPI {
    public function helloWorld() : String {
        return "Hello!"
    }
}
```

ClaimCenter publishes the WSDL for the `HelloWorldAPI` web service at this location.

```
http://localhost:PORTNUMBER/cc/ws/example/HelloWorldAPI?WSDL
```

ClaimCenter generates the following WSDL for the `HelloWorldAPI` web service.

```
<?xml version="1.0"?>
<!-- Generated WSDL for example.HelloWorldAPI web service -->
<wsdl:definitions targetNamespace="http://example.com/example/HelloWorldAPI"
    name="HelloWorldAPI" xmlns="http://example.com/example/HelloWorldAPI"
    xmlns:gw="http://guidewire.com/xsd" xmlns:soap11="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
    <wsdl:types>
        <xss:schema targetNamespace="http://example.com/example/HelloWorldAPI"
            elementFormDefault="qualified" xmlns:xss="http://www.w3.org/2001/XMLSchema">
            <!-- helloWorld() : java.lang.String -->
            <xss:element name="helloWorld">
                <xss:complexType/>
            </xss:element>
            <xss:element name="helloWorldResponse">
                <xss:complexType>
                    <xss:sequence>
                        <xss:element name="return" type="xss:string" minOccurs="0"/>
                    </xss:sequence>
                </xss:complexType>
            </xss:element>
        </xss:schema>
    </wsdl:types>
    <wsdl:portType name="HelloWorldAPIServicePortType">
        <wsdl:operation name="helloWorld">
            <wsdl:input name="helloWorld" message="helloWorld"/>
            <wsdl:output name="helloWorldResponse" message="helloWorldResponse"/>
        </wsdl:operation>
    </wsdl:portType>
    <wsdl:binding name="HelloWorldAPISoap12Binding" type="HelloWorldAPIServicePortType">
        <soap12:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
        <wsdl:operation name="helloWorld">
            <soap12:operation style="document"/>
            <wsdl:input name="helloWorld">
                <soap12:body use="literal"/>
            </wsdl:input>
            <wsdl:output name="helloWorldResponse">
                <soap12:body use="literal"/>
            </wsdl:output>
        </wsdl:operation>
    </wsdl:binding>
    <wsdl:binding name="HelloWorldAPISoap11Binding" type="HelloWorldAPIServicePortType">
        <soap11:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
        <wsdl:operation name="helloWorld">
```

```

<soap11:operation style="document"/>
<wsdl:input name="helloWorld">
  <soap11:body use="literal"/>
</wsdl:input>
<wsdl:output name="helloWorldResponse">
  <soap11:body use="literal"/>
</wsdl:output>
</wsdl:operation>
</wsdl:binding>
<wsdl:service name="HelloWorldAPI">
  <wsdl:port name="HelloWorldAPISoap12Port" binding="HelloWorldAPISoap12Binding">
    <soap12:address location="http://localhost:8180/cc/ws/example/HelloWorldAPI"/>
    <gw:address location="${cc}/ws/example/HelloWorldAPI"/>
  </wsdl:port>
  <wsdl:port name="HelloWorldAPISoap11Port" binding="HelloWorldAPISoap11Binding">
    <soap11:address location="http://localhost:8180/cc/ws/example/HelloWorldAPI"/>
    <gw:address location="${cc}/ws/example/HelloWorldAPI"/>
  </wsdl:port>
</wsdl:service>
<wsdl:message name="helloWorld">
  <wsdl:part name="parameters" element="helloWorld"/>
</wsdl:message>
<wsdl:message name="helloWorldResponse">
  <wsdl:part name="parameters" element="helloWorldResponse"/>
</wsdl:message>
</wsdl:definitions>

```

The preceding generated WSDL defines multiple *ports* for this web service. A *port* in the context of a web service is unrelated to ports in the TCP/IP network transport protocol. Web service ports are alternative versions of a published web service. The preceding WSDL defines a SOAP 1.1 version and a SOAP 1.2 version of the HelloWorldAPI web service.

Calling a ClaimCenter web service from Java

For web services, ClaimCenter does not generate libraries for stub classes for use in Java. You must use your own procedures for converting WSDL for the web service into APIs in your preferred language. The following tools are among those that can create Java classes from the WSDL:

- The `wsimport` tool in the Java API for XML Web Services (JAX-WS)

The topics that demonstrate creating client connections to the ClaimCenter web services use the `wsimport` tool and its output.
- The CXF open source tool
- The Axis2 open source tool

Using the `wsimport` tool to generate Java source and class files

The Java API for XML Web Services (JAX-WS) includes a tool called `wsimport` that generates Java-compatible stubs from a WSDL file.

The `wsimport` tool supports getting a WSDL file that is published over the Internet from a running application. The tool also supports using local WSDL files. Refer to the `wsimport` documentation for details on that functionality.

The directory structure of the files that the `wsimport` tool generates is the root URL of the web service namespace in reverse order of its parts, followed by the rest of the namespace. The namespace is a URL that each published web service specifies in its declaration. The URL represents a namespace for all the objects in the WSDL. A typical namespace specifies a company domain name and other meaningful disambiguating or grouping information about the purpose of the service. For example, <http://mycompany.com/ws/myco>HelloWorld>.

If you use Gosu to write a new web service for ClaimCenter, you specify the namespace for the web service by passing a namespace as a `String` argument to the `@WebService` annotation. If you do not override the namespace when you declare the web service, the default value is <http://example.com>.

For a typical web service, the `wsimport` tool generates multiple supporting classes as well as the web service class.

The path to the Java source files that the `wsimport` tool generates has the following structure.

`CURRENT_DIRECTORY/JAVA_SOURCE_DIRECTORY/REVERSED_URL_ROOT/NAMESPACE/CLASS_NAME.java`

For example, suppose your web service's fully qualified class name is `myco.HelloWorld` and the namespace is `http://mycompany.com/ws/myco/HelloWorld`. If you use the `JAVA_SOURCE_DIRECTORY` value `src`, the `wsimport` tool generates the `.java` file at the following location:

```
CURRENT_DIRECTORY/src/com/mycompany/myco/helloworld/HelloWorld.java
```

If you do not set the namespace in your web service, the default value of `http://example.com` is used to set up the directory structure. In this case, the `wsimport` tool generates the `.java` file at the following location:

```
CURRENT_DIRECTORY/src/com/example/myco/HelloWorld.java
```

As well as the Java source files, the `wsimport` tool generates compiled Java class files.

The path to the Java class files that the `wsimport` tool generates has the following structure:

```
CURRENT_DIRECTORY/REVERSED_URL_ROOT/NAMESPACE/CLASS_NAME.class
```

For example, suppose your web service's fully qualified class name is `myco.HelloWorld` and the namespace is `http://mycompany.com/ws/myco/HelloWorld`. The `wsimport` tool generates the `.class` file at the following location:

```
CURRENT_DIRECTORY/com/mycompany/myco/helloworld>HelloWorld.class
```

Generate Java classes that make SOAP client calls to a SOAP API

Procedure

1. Launch the server that publishes the web services.
2. On the computer from which you will run your SOAP client code, open a command prompt.
3. Change the working directory to a place on your local disk where you want to generate Java source files and `.class` files.
4. Create the subdirectory of the current directory where you want to place the Java source files.

For example, you might choose the folder name `src`. Ensure that the subdirectory exists before you run the `wsimport` tool in the next step. This topic calls this subdirectory `JAVA_SOURCE_DIRECTORY`.

5. Type the following command:

```
wsimport WSDL_LOCATION_URL -s JAVA_SOURCE_DIRECTORY
```

If the `JAVA_SOURCE_DIRECTORY` directory does not already exist, the `wsimport` action fails with the error `directory not found`.

For `WSDL_LOCATION_URL`, type the HTTP path to the WSDL. For example:

```
wsimport http://localhost:8080/cc/ws/example/HelloWorldAPI?WSDL -s src
```

The tool generates Java source files and compiled class files. Typically, you need only the class files or the source files, but not both.

6. Make the either the source or the class files available to your Java project.
 - To use the `.java` files, copy the `JAVA_SOURCE_DIRECTORY` subdirectory into your Java project's `src` folder.
 - To use the `.class` files, copy the files to your Java project's `src` folder or add that directory to your project's class path.
7. Write Java SOAP API client code that compiles against these new generated classes.

Using the generated Java classes

To get a reference to the API itself, access the WSDL port (the service type) with the following syntax.

```
new API_INTERFACE_NAME().getAPI_INTERFACE_NAMESoap11Port();
```

For example, for the API interface name `HelloWorldAPI`, the Java code looks like the following statement.

```
HelloWorldAPIServiceType port = new HelloWorldAPI().getHelloWorldAPISoap11Port();
```

Once you have that port reference, you can call your web service API methods directly on it.

You can publish a web service that does not require authentication by overriding the set of permissions necessary for the web service. See “Specifying required permissions for a web service” on page 77. The following is a simple example to show calling the web service without worrying about the authentication-specific code.

```
import com.example.example.helloworldapi.HelloWorldAPI;
import com.example.example.helloworldapi.HelloWorldAPIServiceType;

public class WsiTestNoAuth {

    public static void main(String[] args) throws Exception {

        // Get a reference to the SOAP 1.1 port for this web service
        HelloWorldAPIServiceType port = new HelloWorldAPI().getHelloWorldAPISoap11Port();

        // Call API methods on the web service
        String res = port.helloWorld();

        // Print result
        System.out.println("Web service result = " + res);
    }
}
```

Adding HTTP basic authentication in Java

In previous topics, the examples of calling a web service from Java show how to connect to a service without authentication. The following code shows how to add HTTP Basic authentication to your client request. HTTP Basic authentication is the easiest type of authentication to code to connect to a ClaimCenter web service.

```
import com.example.example.helloworldapi.HelloWorldAPI;
import com.example.example.helloworldapi.HelloWorldAPIServiceType;
import com.sun.xml.internal.ws.api.message.Headers;
import javax.xml.ws.BindingProvider;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import javax.xml.namespace.QName;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.ws.BindingProvider;
import java.util.Map;

public class WsiTest02 {

    public static void main(String[] args) throws Exception {
        System.out.println("Starting the web service client test...");

        // Get a reference to the SOAP 1.1 port for this web service
        HelloWorldAPIServiceType port = new HelloWorldAPI().getHelloWorldAPISoap11Port();

        // Cast to BindingProvider so the following lines are easier to understand
        BindingProvider bp = (BindingProvider) port;

        // "HTTP Basic" authentication
        Map<String, Object> requestContext = bp.getRequestContext();
        requestContext.put(BindingProvider.USERNAME_PROPERTY, "su");
        requestContext.put(BindingProvider.PASSWORD_PROPERTY, "gw");

        System.out.println("Calling the service now...");
        String res = port.helloWorld();
        System.out.println("Web service result = " + res);
    }
}
```

Adding SOAP header authentication in Java

Although HTTP authentication is the easiest to code for most integration programmers, ClaimCenter also supports optionally authenticating web services using custom SOAP headers.

For Guidewire applications, the structure of the required SOAP header contains the following elements.

- An <authentication> element with the namespace `http://guidewire.com/ws/soapheaders`.
That element contains two elements.
 - <username> – Contains the username text
 - <password> – Contains the password text

This SOAP header authentication option is also known as Guidewire authentication.

The Guidewire authentication XML element looks like the following.

```
<?xml version="1.0" encoding="UTF-16"?>
<authentication xmlns="http://guidewire.com/ws/soapheaders"><username>su</username><password>gw
</password></authentication>
```

The following code shows how to use Java client code to access a web service with the fully-qualified name `example.helloworldapi.HelloWorld` using a custom SOAP header.

After authenticating, the example calls the `helloWorld` SOAP API method.

```
import com.example.example.helloworldapi.HelloWorldAPI;
import com.example.example.helloworldapi.HelloWorldAPI;
import com.example.example.helloworldapi.HelloWorldAPIPortType;
import com.sun.org.apache.xml.internal.serialize.DOMSerializerImpl;
import com.sun.xml.internal.messaging.saaj.soap.ver1_1.Header1_1Impl;
import com.sun.xml.internal.messaging.saaj.soap.ver1_1.HeaderElement1_1Impl;
import com.sun.xml.internal.ws.developer.WSBindingProvider;
import com.sun.xml.internal.ws.api.message.Headers;
import org.w3c.dom.Document;
import org.w3c.dom.Element;

import javax.xml.namespace.QName;
import javax.xml.parsers.DocumentBuilderFactory;

public class WsITest01 {

    private static final QName AUTH = new QName("http://guidewire.com/ws/soapheaders",
        "authentication");
    private static final QName USERNAME = new QName("http://guidewire.com/ws/soapheaders", "username");
    private static final QName PASSWORD = new QName("http://guidewire.com/ws/soapheaders", "password");

    public static void main(String[] args) throws Exception {
        System.out.println("Starting the web service client test...");

        // Get a reference to the SOAP 1.1 port for this web service.
        HelloWorldAPIPortType port = new HelloWorldAPI().getHelloWorldAPISoap11Port();

        // Cast to WSBindingProvider so the following lines are easier to understand
        WSBindingProvider bp = (WSBindingProvider) port;

        // Create XML for special SOAP headers for Guidewire authentication of a user & password.
        Document doc = DocumentBuilderFactory.newInstance().newDocumentBuilder().newDocument();
        Element authElement = doc.createElementNS(AUTH.getNamespaceURI(), AUTH.getLocalPart());
        Element usernameElement = doc.createElementNS(USERNAME.getNamespaceURI(),
            USERNAME.getLocalPart());
        Element passwordElement = doc.createElementNS(PASSWORD.getNamespaceURI(),
            PASSWORD.getLocalPart());

        // Set the username and password, which are content within the username and password elements.
        usernameElement.setTextContent("su");
        passwordElement.setTextContent("gw");

        // Add the username and password elements to the "Authentication" element.
        authElement.appendChild(usernameElement);
        authElement.appendChild(passwordElement);

        // Uncomment the following lines to see the XML for the authentication header.
        DOMSerializerImpl ser = new DOMSerializerImpl();
        System.out.println(ser.writeToString(authElement));

        // Add the authentication element to the list of SOAP headers.
        bp.setOutboundHeaders(Headers.create(authElement));
    }
}
```

```
System.out.println("Calling the service now...");
String res = port.helloWorld();
System.out.println("Web service result = " + res);
}
```

Testing web services with local WSDL

For testing purposes only, you can call web services published from the same ClaimCenter server. To call a web service on the same server, you must generate WSDL files into the class file hierarchy so Gosu can access the service. This permits you to write unit tests that access the WSDL files over the SOAP protocol from Gosu.

Guidewire does not support calls to SOAP APIs published on the same server in a production system.

Generating WSDL for local web services

From the ClaimCenter installation directory, run the following command.

```
gwb genWsiLocal
```

ClaimCenter generates the WSDL in the `wsi.local` package, followed by the fully qualified name of the web service.

```
wsi.local.FQNAME.wsdl
```

Using your class to test local web services

To use the class, prefix the text `wsi.local.` (with a final period) to the fully-qualified name of your API implementation class.

For example, suppose the web service implementation class is:

```
mycompany.ws.v100.EchoAPI
```

The command `gwb genWsiLocal` generates the following WSDL file.

```
ClaimCenter/modules/configuration/gsrc/wsi/local/mycompany/ws/v100/EchoAPI.wsdl
```

Call this web service locally with the following line of code.

```
var api = new wsi.local.mycompany.ws.v100.EchoAPI()
```

If you change the code for the web service and the change potentially changes the WSDL, regenerate the WSDL.

ClaimCenter includes WSDL for some local services in the base configuration. These WSDL files are in Studio modules other than `configuration`. If ClaimCenter creates new local WSDL files from the `gwb genWsiLocal` tool, it creates new files in the `configuration` module in the `gsrc` directory.

The `wsi.local.*` namespace exists only to call web services from GUnit unit tests. It is unsafe to write production code that ever uses or calls `wsi.local.*` types.

To reduce the chance of accidental use of `wsi.local.*` types, Gosu prevents using these types in method signatures of published web services.

Writing unit tests for your web service

It is good practice to design your web services to be testable. At the time you design your web service, think about the kinds of data and commands your service handles. Consider your assumptions about the arguments to your web service, what use cases your web service handles, and which use cases you want to test. Then, write a series of GUnit tests that use the `wsi.local.*` namespace for your web service.

For example, you created the following web service.

```
package example
uses gw.xml.ws.annotation.WsiWebService
@WsiWebService("http://mycompany.com")
class HelloWorldAPI {
    public function helloWorld() : String {
        return "Hello!"
    }
}
```

The following sample Gosu code is a GUnit test of the preceding `HelloWorldAPI` web service.

```
package example
uses gw.testharness.TestBase
class HelloWorldAPITest extends gw.testharness.TestBase {
    construct() {
    }

    public function testMyAPI() {
        var api = new wsi.local.example.helloworldapi.HelloWorldAPI()
        api.Config.Guidewire.Authentication.Username = "su"
        api.Config.Guidewire.Authentication.Password = "gw"
        var res = api.helloWorld();
        print("result is: " + res);
        TestBase.assertEquals("Expected 'Hello!'", "Hello!", res)
        print("we got to the end of the test without exceptions!")
    }
}
```

For more thorough testing, test your web service from integration code on an external system. To ensure your web service scales adequately, test your web service with as large a data set and as many objects as potentially exist in your production system. To ensure the correctness of database transactions, test your web service to exercise all bundle-related code.

Web services authentication plugin

To handle the name/password authentication for a user connecting to web services, ClaimCenter delegates the task to the currently registered implementation of the `WebservicesAuthenticationPlugin` plugin interface. There must always be a registered version of this plugin, otherwise web services that require permissions cannot authenticate successfully.

The `WebservicesAuthenticationPlugin` plugin interface supports web service connections only.

Web services authentication plugin implementation

To authenticate web service requests, a common practice is to create fictional ClaimCenter users whose express purpose is to provide authentication. You can create fictional users by using the ClaimCenter user administration feature. An external system making a web service request is authenticated by passing the fictional user's name and password in the service's request headers.

The base configuration of ClaimCenter includes a registered implementation of the web service authentication plugin. The class is called `DefaultWebservicesAuthenticationPlugin`. The class performs two main operations:

1. Scans HTTP request headers for authentication information.
2. Performs authentication against the local ClaimCenter users in the database. The class calls the registered implementation of the `AuthenticationServicePlugin` plugin interface.

If you write your own implementation of the `AuthenticationServicePlugin` plugin interface, be aware of the interaction with web service authentication. For example, you might want LDAP authentication for most users, but for

web service authentication, you would want to authenticate against the current ClaimCenter application administrative data.

To authenticate only some user names to Guidewire application credentials, your `AuthenticationServicePlugin` code must check the user name and compare to a list of special web service user names. If the user name matches, do not use LDAP, but instead authenticate with the local application administrative data. To do this authentication in your `AuthenticationServicePlugin` implementation, use the following code.

```
_handler.verifyInternalCredentials(username, password)
```

See also

- “User authentication service plugin” on page 191

Writing an implementation of the web services authentication plugin

Most environments do not require writing a new implementation of the `WebservicesAuthenticationPlugin` plugin interface. Typical changes to authentication logic are instead in your `AuthenticationServicePlugin` plugin implementation.

You can change the default web services authentication behavior to get the credentials from different headers. You can write your own `WebservicesAuthenticationPlugin` plugin implementation to implement custom logic.

The `WebservicesAuthenticationPlugin` interface has a single method that your implementation must implement called `authenticate`. The `authenticate` method accepts a parameter of type `WebservicesAuthenticationContext`. The object passed to the `authenticate` method contains authentication information, such as the name and password.

The relevant properties in the `WebservicesAuthenticationContext` argument are listed below.

- `HttpHeaders` – HTTP headers of type `gw.xml.ws.HttpHeaders`. The argument includes a list of header names.
- `RequestSOAPHeaders` – The request SOAP headers as an `XmlElement` object.

If authentication succeeds, the `authenticate` method returns the relevant `User` object from the ClaimCenter database. If authentication could not be attempted, such as if network problems existed, the method returns `null`. Other causes of authentication failure throw a `WsAuthenticException`.

Login authentication confirmation

In typical cases, web service client code sets up authentication and calls web services, relying on catching any exceptions if authentication fails. You do not need to call a specific web service as a precondition for login authentication. In effect, authentication happens with each API call.

However, if you want your web service client code to explicitly test specific authentication credentials, ClaimCenter publishes the built-in `Login` web service. Call this web service’s `login` method, which takes a user name as a `String` and a password as a `String`. If authentication fails, the API throws an exception.

If the authentication succeeds, that server creates a persistent server session for that user ID and returns the session ID as a `String`. The session persists after the call completes. In contrast, a normal API call creates a server session for that user ID but clears the session as soon as the API call completes.

If you call the `login` method, it is a good practice to call the matching `logout` method to clear the session, passing the session ID as an argument. If you are merely trying to confirm authentication, you can call `logout` immediately.

However, in some rare cases you might want to leave the session open for logging purposes to track the owner of multiple API calls from one external system. After you complete your multiple API calls, finally call `logout` with the original session ID.

If you do not call `logout`, all application servers are configured to time out the session eventually.

Request or response XML structural transformations

For advanced layers of security, you probably want to use transformations that use the byte stream. However, there are other situations where you might want to transform either the request or response XML data at the structural level of manipulating `XmlElement` objects.

To transform the request envelope XML before processing, add the `@WsiRequestXmlTransform` annotation. To transform the response envelope XML after generating it, add the `@WsiResponseXmlTransform` annotation.

Each annotation takes a single constructor argument which is a Gosu block. Pass a block that takes one argument, which is the envelope. The block transforms the XML element in place using the Gosu XML APIs.

The envelope reference statically has the type `XmlElement`. However, at runtime the type is one of the following, depending on whether SOAP 1.1 or SOAP 1.2 invoked the service.

- `gw.xsd.w3c.soap11_envelope.Envelope`
- `gw.xsd.w3c.soap12_envelope.Envelop`

See also

- “Transformations on data streams” on page 100

Transforming a generated schema

ClaimCenter generates WSDL and XSDs for the web service based on the contents of your implementation class and any of its configuration-related annotations. In typical cases, the generated files are appropriate for consuming by any web service client code.

In rare cases, you might need to do some transformation. For example, if you want to specially mark certain fields as required in the XSD itself, or to add other special comment or marker information.

You can do any arbitrary transformation on the schema by adding the `@WsiSchemaTransform` annotation. The one argument to the annotation constructor is a block that does the transformation. The block takes two arguments.

- A reference to the entire WSDL XML. From Gosu, this object is an `XmlElement` object and strongly typed to match the `<definitions>` element from the official WSDL XSD specification. From Gosu, this type is `gw.xsd.w3c.wsdl.Definitions`.
- A reference to the main generated schema for the operations and its types. From Gosu this object is an `XmlElement` object strongly typed to match the `<schema>` element from the official XSD metaschema. From, Gosu this type is `gw.xsd.w3c.xmlschema.Schema`. There may be additional schemas in the WSDL that are unrepresented by this parameter but are accessible through the WSDL parameter.

The following example modifies a schema to force a specific field to be required. In other words, it strictly prohibits a null value. This example transformation finds a specific field and changes its XSD attribute `MinOccurs` to be 1 instead of 0.

```
@WsiSchemaTransform( \ wsdl, schema ->{  
    schema.Element.firstWhere( \ e ->e.Name == "myMethodSecondParameterIsRequired"  
        ).ComplexType.Sequence.Element[1].MinOccurs = 1  
} )
```

You can also change the XML associated with the WSDL outside the schema element.

Converting a Gosu object to and from XML

When passing or returning a Gosu object in a web service, the object often needs to be converted to and from XML. Converting a Gosu object to XML is called *marshalling*. Similarly, converting XML back to a Gosu object is called *unmarshalling*.

Converting a Gosu object to XML

The `marshal` method in the `gw.xml.ws.WsiExportableUtil` class converts a Gosu DTO and any other marshallable object to XML.

```
public static function marshal(element : XmlElement, obj : Object)
```

The method accepts two parameters.

element

Stores the generated XML

obj

An instance of a marshallable Gosu class

The following types are marshallable.

- Simple types, such as `int`, `Integer`, `Double`, `String`, `boolean`, and `Boolean`
- `GWRemotable` types
- GX models
- XSD types
- Lists of marshallable types

The `marshal` method has no return value. The generated XML is stored in the `element` argument. If the object is not marshallable, an `IllegalArgumentException` is thrown.

If the `obj` argument is a Gosu DTO and a namespace is specified in its `@WsiExportable` annotation, the same namespace is used to contain the generated XML. Otherwise, a namespace is generated based on the object's package name.

For every public property with a non-null value in the `obj` argument, the conversion process defines an XML element using the property's name. A public property with a `null` value is not converted and is not referenced in the generated XML.

To convert the generated XML to a series of raw bytes, call the `bytes` method of the `XmlElement` object. To convert the XML to a `String`, call the object's `asUTFString` method.

The following Gosu class definition meets the requirements of a DTO. Notice that a namespace is not specified in the `@WsiExportable` annotation. The resulting XML will use a generated namespace based on the object's package name.

```
package gw.examples

uses gw.xml.ws.annotation.WsiExportable

@WsiExportable
final class Car {
    // Private fields defined with public Gosu property names in the recommended style
    private var _color : String as Color
    private var _model : String as Model

    // Override toString() so it can be passed to the print() method for testing
    override function toString() : String {
        return "Car: Color=${_color} Model=${_model}"
    }
}
```

The following code marshals an instance of the Gosu DTO class.

```
uses gw.xml.ws.WsiExportableUtil
uses gw.xml.XmlElement

// Instantiate a DTO object
var obj = new gw.examples.Car()
obj.Color = "Blue"
obj.Model = "234"

// Create an XML element
var xml = new XmlElement("root")

// Convert the DTO object to XML
```

```
WsiExportableUtil.marshal(xml, obj)
// Verify the resulting XML by converting it to a String
var xmlString = xml.asUTFString()
print(xmlString)
```

The example code prints the following output.

```
<?xml version="1.0"?>
<root xmlns:pogo="http://example.com/gw/examples">
  <pogo:Color>Blue</pogo:Color>
  <pogo:Model>234</pogo:Model>
</root>
```

The pogo: prefix is the short name for the namespace that was generated by the `marshal` method. If the `@WsiExportable` annotation had specified an optional `String` namespace argument, that namespace would have been used in the generated XML.

Converting XML to a Gosu object

The `unmarshal` method in the `gw.xml.ws.WsiExportableUtil` class converts XML back to a Gosu DTO or any other marshallable object.

```
public static function unmarshal(element : XmlElement, IType : type) : Object
```

The method accepts two parameters.

element

Contains the XML to convert

The data must match the XML structure generated by the `marshal` method for the specified Gosu object type.

type

Specifies the type of the marshallable Gosu object

If the object cannot be unmarshalled, a `RuntimeException` is thrown.

The `unmarshal` method returns a base `Object` initialized using the converted XML values. The returned base object must be downcast to the required marshallable Gosu type.

Alternatively, the required Gosu type can be specified when calling the `unmarshal` method by using Gosu generics syntax. In this case, the method accepts only the `element` argument and returns a Gosu object of the required type.

The following code converts XML to a Gosu DTO.

```
uses gw.xml.ws.WsiExportableUtil
uses gw.xml.XmlElement

// Initialize a new XmlElement by parsing XML specified in a String
var incomingXML : XmlElement
incomingXML = XmlElement.parse(xmlString)

// Convert the XmlElement into an instance of Gosu DTO class gw.examples.Car
// Use generics syntax to specify the required Gosu DTO type
var car2 = unmarshal<gw.examples.Car>(incomingXML)

// Alternatively, specify the DTO type as an argument and downcast the return value
// var car2 = unmarshal(incomingXML, gw.examples.Car) as gw.examples.Car

// Verify the results by implicitly calling the object's toString() method
print(car2)
```

The example code prints the following output.

```
Car: Color=Blue Model=234
```

Using predefined XSD and WSDL

In typical ClaimCenter implementations, a web service is implemented by a Gosu class with each method implementing one web service operation. ClaimCenter generates appropriate WSDL for all operations in the web service. For any method arguments and return types, Gosu uses the class definition and the method signatures to determine the structure of the WSDL.

Some organizations require web service publishers to conform to predefined XML types defined in an XSD, or even predefine WSDL. However, if you can use shared XSDs instead of a predefined WSDL, it is recommended to use shared XSDs only.

You can write your web service to use a standard XSD to define individual types. For example, suppose a pre-existing WSDL defines an XSD that defines 200 object types for method arguments and return types. You can add the XSD to the source code tree in Studio and access all the XSD types directly from Gosu in a type-safe way. In this case, the XSD types are acting as Data Transfer Objects (DTOs), a general term for a constrained data definition specifically for web services. See the cross references at the end of this section.

If you are required to implement a preexisting service definition in a specific predefined WSDL, you can do so using a feature called invocation handlers. By using a standardized WSDL, an organization can ensure that all related systems conform to predefined specifications. The technical approach of invocation handlers can lead to more complexity in your Gosu code, which can make it harder to catch problems at compile time.

Adding an invocation handler for preexisting WSDL

Only use the `@WsiInvocationHandler` annotation if you need to write a web service that conforms to externally defined standard WSDL. Using this approach makes your code harder to read and also error prone because mistakes are harder to catch at compile time. For example, it is harder to catch errors in return types using this approach.

To implement preexisting WSDL, you define your web service very differently than for typical web service implementation classes.

First, on your web service implementation class add the annotation `@WsiInvocationHandler`. As an argument to this annotation, pass an invocation handler. An invocation handler has the following qualities.

- The invocation handler is an instance of a class that extends the type `gw.xml.ws.annotation.DefaultWsiInvocationHandler`.
- Implement the invocation handler as an inner class inside your web service implementation class.
- The invocation handler class overrides the `invoke` method with the following signature.

```
override function invoke(requestElement : XmlElement, context : WsiInvocationContext) : XmlElement
```

- The `invoke` method does the actual dispatch work of the web service for all operations on the web service. Gosu does not call any other methods on the web service implementation. Instead, the invocation handler handles all operations that normally would be in multiple methods on a typical web service implementation class.
- Your `invoke` method can call its super implementation to trigger standard method calls for each operation based on the name of the operation. Use this technique to run custom code before or after standard method invocation, either for logging or special logic.

In the `invoke` method of your invocation handler, determine which operation to handle by checking the type of the `requestElement` method parameter. For each operation, perform whatever logic makes sense for your web service. Return an object of the appropriate type. Get the type of the return object from the XSD-based types created from the WSDL.

Finally, for the WSDL for the service to generate successfully, add the preexisting WSDL to your web service using the parse options annotation `@WsiParseOptions`. Pass the entire WSDL as the schema as described in that topic.

Example of an invocation handler for preexisting WSDL

In the following example, there is a WSDL file at the resource path in the source code tree at the path.

```
ClaimCenter/configuration/gsrc/ws/weather.wsdl
```

The schema for this file has the Gosu syntax: `ws.weather.util.SchemaAccess`. Its element types are available in the Gosu type system as objects in the package `ws.weather.elements`.

The method signature of the `invoke` method returns an object of type `XmlElement`, the base class for all XML elements. Be sure to carefully create the right subtype of `XmlElement` that appropriately corresponds to the return type for every operation.

The following example implements a web service that conforms to a preexisting WSDL and implements one of its operations.

```
package gw.xml.ws

uses gw.xml.ws.annotation.WsiWebService
uses gw.xml.ws.annotation.WsiInvocationHandler
uses gw.xml.XmlElement
uses gw.xml.ws.annotation.WsiPermissions
uses gw.xml.ws.annotation.WsiAvailability
uses gw.xml.ws.annotation.WsiParseOptions
uses gw.xml.XmlParseOptions
uses java.lang.IllegalArgumentException

@WsiWebService("http://guidewire.com/px/ws/gw/xml/ws/WsiImplementExistingWsdlTestService")
@WsiPermissions({})
@WsiAvailability(NONE)
@WsiInvocationHandler(new WsiImplementExistingWsdlTestService.Handler())
@WsiParseOptions(new XmlParseOptions() { :AdditionalSchemas = { ws.weather.util.SchemaAccess } })

class WsiImplementExistingWsdlTestService {

    // The following line declares an INNER CLASS within the outer class.
    static class Handler extends DefaultWsiInvocationHandler {

        // Here we implement the "weather" wsdl with our own GetCityForecastByZIP implementation.
        override function invoke(requestElement : XmlElement, context : WsiInvocationContext)
            : XmlElement {

            // Check the operation name. If it is GetCityForecastByZIP, handle that operation.
            if (requestElement typeis ws.weather.elements.GetCityForecastByZIP) {
                var returnResult = new ws.weather.elements.GetCityForecastByZIPResponse() {

                    // The next line uses type inference to instantiate XML object of the correct type
                    // rather than specifying it explicitly.
                    :GetCityForecastByZIPResult = new() {
                        :Success = true,
                        :City = "Demo city name for ZIP ${requestElement.ZIP}"
                    }
                    return returnResult
                }
            }

            // Check for additional requestElement values to handle additional operations.
            if ...

            else {
                throw new IllegalArgumentException("Unrecognized element: " + requestElement)
            }
        }
    }
}
```

First, the `invoke` method checks if the requested operation is the desired operation. An operation normally corresponding to a method name on the web service, but in this approach one method handles all operations. In this simple example, the `invoke` method handles only the operation in the WSDL called `GetCityForecastByZip`. If the requested operation is `GetCityForecastByZip`, the code creates an instance of the `GetCityForecastByZIPResponse` XML element.

Next, the example uses Gosu object creation initialization syntax to set properties on the element as appropriate. Finally, the code returns that XML object to the caller as the result from the API call.

For additional context of the WSI request, use the `context` parameter to the `invoke` method. The `context` parameter has type `WsiInvocationContext`, which contains properties such as servlet and request headers.

Invocation handler responsibilities

If you write an invocation handler for a web service, by default you are bypassing some important features.

- The application does not enable profiling for method calls.
- The application does not check run levels even at the web service class level.
- The application does not check web service permissions, even at the web service class level.
- The application does not check for duplicate external transaction IDs if present.

However, you can support all these things in your web service even when using an invocation handler, and in typical cases it is best to do so.

Re-enable bypassed features from an invocation handler

Procedure

1. In your web service implementation class, create separate methods for each web service operation. For each method, for the one argument and the one return value, use an `XmlElement` object.

```
static function myMethod(req : XmlElement) : XmlElement
```
2. In your invocation handler's `invoke` method, determine which method to call based on the operation name, as documented earlier.
3. Get a reference to the method info meta data for the method you want to call, using the `#` symbol to access meta data of a feature (method or property).

```
var MethodInfo = YourClassName#myMethod(XmlElement).MethodInfo
```
4. Before calling your method, get a reference to the `WsiInvocationContext` object that is a method argument to `invoke`. Call its `preExecute` method, passing the `requestElement` and the method info metadata as arguments. If you do not require checking method-specific annotations for run level or permissions, for the method info metadata argument you can pass `null`.

The `preExecute` method does several things.

- Enables profiling for the method you are about to call if profiling is available and configured.
 - Checks the SOAP headers looking for headers that set the locale. If found, sets the specified locale.
 - Checks the SOAP headers for a unique transaction ID. This ID is intended to prevent duplicate requests. If this transaction has already been processed successfully (a bundle was committed with the same ID), `preExecute` throws an exception.
 - If the method info argument is non-null, `preExecute` confirms the run level for this service, checking both the class level `@WsiAvailability` annotation and any overrides for the method. As with standard implementation classes, the method level annotation supersedes the class level annotation. If the run level is not at the required level, `preExecute` throws an exception.
 - If the method info argument is non-null, `preExecute` confirms user permissions for this service, checking both the class level `@WsiPermissions` annotation and any overrides for the method. As with standard implementation classes, the method level annotation supersedes the class level annotation. If the permissions are not satisfied for the web service user, `preExecute` throws an exception.
5. Call your method from the invocation handler.

What to do next

See also

- “Example of an invocation handler for preexisting WSDL” on page 93

Checking the method-specific annotations for run level or permissions

To check the method-specific annotations for run level or permissions, one approach is to set up a map to store the method information. The map key is the operation name. The map value is the method info metadata required by the `preExecute` method.

The following example demonstrates this approach.

```
@WsiInvocationHandler( new WsiImplementExistingWsdlTestService.Handler())
class WsiImplementExistingWsdlTestService {

    // The following line declares an INNER class within the outer class.
    static class Handler extends DefaultWsiInvocationHandler {

        var _map = { "myOperationName1" -> WsiImplementExistingWsdlTestService#methodA(XmlElement).MethodInfo,
                    "myOperationName2" -> WsiImplementExistingWsdlTestService#methodB(XmlElement).MethodInfo
                }

        override function invoke( requestElement : XmlElement, context : WsiInvocationContext )
            : XmlElement {
            // Get the operation name from the request element
            var opName = requestElement.QName

            // Get the local part (short name) from operation name, and get the method info for it
            var method = _map.get(opName.LocalPart)

            // Call preExecute to enable some features otherwise disabled in an invocation handler
            context.preExecute(requestElement, method)

            // Call your method using the method info and return its result
            return method.CallHandler.handleCall(null, {requestElement}) as XmlElement
        }

        // After defining your invocation handler inner class, define the methods that do your work
        // as separate static methods

        // Example of overriding the default permissions
        @WsiPermissions( { /* add a list of permissions here */ } )
        static function methodA(req : XmlElement) : XmlElement {
            /* Do whatever you do, and return the result */
            return null
        }
        static function methodB(req : XmlElement) : XmlElement {
            /* do whatever you do, and return the result */
            return null
        }
    }
}
```

This use is the only supported one for a `WsiInvocationContext` object's `preExecute` method. Any use other than calling it exactly once from within an invocation handler `invoke` method is unsupported.

Referencing additional schemas in your published WSDL

If you need to expose additional schemas to the web service clients in the WSDL, you can use the `@WsiAdditionalSchemas` annotation. Use this annotation to provide references to schemas that might be required but are not automatically included.

For example, you might define an operation to take any object in a special situation, but actually accept only one of several different elements defined in other schemas. You might throw exceptions on any other types. By using this annotation, the web service can add specific new schemas so that web service client code can access them from the WSDL for the service.

The annotation takes one argument of the type `List<XmlSchemaAccess>`, which is a list of schema access objects. To get a reference to a schema access object, first put an XSD in your class hierarchy. Then from Gosu, determine the fully qualified name of the XSD based on where you put the XSD. Next, get the `util` property from the schema, and on the result get the `SchemaAccess` property. To generate a list, surround one or more items with curly braces and comma-separate the list.

For example, the following annotation adds the XSD that resides locally in the location `gw.xml.ws.wsimyschema.util`:

```
@WsiAdditionalSchemas({ gw.xml.ws.wsimyschema.util.SchemaAccess })
```

Validating requests using additional schemas as parse options

You can validate incoming requests by using additional schemas. To add an additional schema parse option, add the `@WsiParseOptions` annotation to your web service implementation class.

Before proceeding, be sure you have a reference to the XSD-based schema. For an XSD or WSDL, get the `SchemaAccess` property on the XSD type to get the schema reference. The argument for the annotation is an instance of type `XmlParseOptions`, which contains a property called `AdditionalSchemas`. That property must contain a list of schemas.

To add a single schema, you can use the following compact syntax.

```
@WsiParseOptions(new XmlParseOptions() { :AdditionalSchemas = { YOUR_XSD_TYPE.util.SchemaAccess } })
```

For an XSD called `ws.xsd` in the source code file tree in the package `com.abc`, use the following syntax.

```
@WsiParseOptions(new XmlParseOptions() { :AdditionalSchemas = { com.abc.ws.util.SchemaAccess } })
```

To include an entire WSDL file as an XSD, use the same syntax. For example, if the file is `WS.wsdl`.

```
@WsiParseOptions(new XmlParseOptions() { :AdditionalSchemas = { com.abc.ws.util.SchemaAccess } })
```

Creating an XML Schema JAR file

After an integration project is finished and stable, the XML and XSD schema files that are created during WSDL code generation rarely change. Instead of having each build recreate these static files, the files can be created a single time and stored in a Schema JAR file. This JAR file is made available to all developers for general access. Subsequent builds use the contents of the preprocessed JAR file to reduce build times.

The Schema JAR file can contain the following types of resources.

- XML and XSD schema files created during WSDL code generation
- WSDL and XSD resources retrieved from remote servers by using Web Service Collections.
- Third-party XML and XSD schema files

The Schema JAR file cannot contain the XML and XSD schema files associated with a GX model.

The Schema JAR file can be created from the command line by running the `gwb` command with the `genSchemaJar` task.

```
gwb genSchemaJar
```

Unversioned and versioned Schema JAR file usage

One way to use the Schema JAR file is to make it available in source control as an unversioned file. In such a situation, integration developers who edit the JAR file's contents also coordinate the JAR builds. They check into source control the built JAR file and its individual component files. The non-integration developers never build the JAR file themselves. Instead, they pull the latest JAR file from source control.

Another way to use the Schema JAR file is to make a new JAR file for each change in its contents. To distinguish the versions, the version string is included in the JAR file name. Integration developers are responsible for generating the JAR file and including the appropriate version ID in its name. Non-integration developers retrieve the required version of the JAR file when they pull it from source control.

Enable and configure the genSchemaJar task

By default, the `genSchemaJar` task is not enabled in the `gwb` command. Perform the following steps to enable the task.

1. In the `settings.gradle` file located in the ClaimCenter installation directory, uncomment the line to include the Schemas module.
2. In the `build.gradle` file located in the `modules/configuration` directory, uncomment the block of lines that supports the Schema module.
3. In the `gwxmlmodule.xml` file located in the `modules/configuration/res` directory, uncomment the dependency line that supports the `genSchemaJar` task.

The heap memory usage for the `genSchemaJar` task can be configured by setting the following values in the `gradle.properties` file. The `gradle.properties` file is located in the ClaimCenter installation directory.

- `custDistJavaCompileSchemaSourcesMinHeapSize` – Minimum JVM heap size when performing the `genSchemaJar` task. Default is eight gigabytes.
- `custDistJavaCompileSchemaSourcesMaxHeapSize` – Maximum JVM heap size when performing the `genSchemaJar` task. Default is 16 gigabytes.

Create the Schema JAR file

Perform the following steps to create the Schema JAR file.

1. Move the XML, XSD schema, and other resource files to include in the JAR file to a directory of your choice in the `modules/schemas/gsrc` tree hierarchy.
2. Create the Schema JAR file by running the `gwb genSchemaJar` task.

```
gwb genSchemaJar
```

The Schema JAR file is a dependency for building the Configuration module. The Configuration module is built by running the `gwb compile` task.

```
gwb compile
```

Modify the Schema JAR file contents

If any file stored in the Schema JAR file requires editing, the following workflow must be performed.

1. Before editing a file stored in the Schema JAR file, move the file back to its original location in the Configuration module.
2. Recreate the Schema JAR file without the moved file.
3. Rebuild the Configuration module using the recreated Schema JAR file.

The file can now be edited.

When all changes are complete, the edited file can be restored to the JAR file by moving it back to the JAR directory. Then rebuild the JAR file and Configuration module.

Support versioned Schema JAR file usage

Optionally, you can add support in the build process for a versioned Schema JAR file.

Edit the following files.

- In the `build.gradle` file located in the ClaimCenter installation directory, append the following lines to the end of the file.

```
ext {  
    xmlSchemaVersion = "1.0.0"  
    xmlSchemaJarFilename = "gw-xml-schemas-${xmlSchemaVersion}.jar".toString()  
}
```

- In the `build.gradle` file located in the `modules/schemas` directory, append the following lines to the end of the file.

```
afterEvaluate {
    tasks.genSchemaJar.conventionMapping.archiveName = { xmlSchemaJarFilename }
}
```

- In the `build.gradle` file located in the `modules/configuration` directory, modify the `genSchemaJar` dependencies block to reference the versioned file name, as shown below.

```
dependencies {
    task (':modules:schemas:genSchemaJar')
    compile files(xmlSchemaJarFilename)
    schemajars files(xmlSchemaJarFilename)
}
```

Whenever the JAR version changes, the Studio project must be regenerated to retrieve the new dependency for the Configuration module. Regenerate the Studio project by running `gwb idea` on the command line. Alternatively, regenerate the project in Studio by selecting **Tools > Generate Project**. Afterward, create the Schema JAR by running `gwb genSchemaJar`. Finally, build the application by either running `gwb compile` on the command line or selecting **Build > Rebuild Project** in Studio.

Publish versioned Schema JAR file to artifact repository

The Schema JAR file can be versioned and published to an artifact repository. Integration developers handle the versioning and publishing operations. Non-integration developers retrieve from source control the latest files that specify the required version. The next build pulls that version from the repository.

In the following example, support is added to the build process to create and publish a versioned Schema JAR file. The version string is not included as part of the JAR file name. The JAR is published to a Maven repository.

- In the `build.gradle` file located in the ClaimCenter installation directory, append the following lines to the end of the file.

```
ext {
    xmlSchemaVersion = "1.0.0"
    xmlSchemaGroupId = 'com.myinsuranceco.cc'
    xmlSchemaArtifactId = 'schemas'
    xmlSchemaJarFilename = 'gw-xml-schemas.jar".toString()
}
```

- In the `build.gradle` file located in the `modules/schemas` directory, add the following line after the existing `apply plugin` lines.

```
apply plugin: 'maven-publish'
```

- In the same file, add the following lines after the closure of the `tasks.compileSchemaSources.options.with` block. Replace the Maven repository `url` reference in the example with the URL of your own repository.

```
group = xmlSchemaGroupId
version = xmlSchemaVersion
artifacts {
    archives genSchemaJar
}
publishing {
    publications {
        maven(MavenPublication) {
            artifact genSchemaJar
        }
    }
    repositories {
        maven { url 'http://nexus.myinsuranceco.com/content/repositories/releases' }
    }
}
```

- In the `build.gradle` file located in the `modules/configuration` directory, add the following lines after the closure of the `genSchemaJar` dependencies block. Replace the Maven repository `url` reference in the example with the URL of your own repository.

```
repositories {
    maven { url 'http://nexus.myinsuranceco.com/content/repositories/releases' }
    mavenLocal() // Used by integration developers only.
}
```

```
def xmlSchemasArtifact = "${xmlSchemaGroupId}:${xmlSchemaArtifactId}:${xmlSchemaVersion}".toString()
dependencies {
    compile(xmlSchemasArtifact)
    schemajars(xmlSchemasArtifact)
}
```

- If the referenced Maven url repository is not a local repository used for files under development, perform the following step.
 - Edit the gwb and gwb.bat files located in the ClaimCenter installation directory to remove all references to the --offline argument.

To publish a new JAR version, perform the following steps.

1. Regenerate the Studio project by running gwb idea.
2. Rebuild the Schema JAR file by running gwb genSchemaJar.
3. Publish the JAR to the local Maven repository by running gwb modules:schemas:publishToMavenLocal. During development of the JAR, continue publishing it to this local repository.
4. Rebuild the application using the published JAR by running gwb compile.
5. When the Schema JAR is ready to be released, publish it to the main Maven repository by running gwb modules:schemas:publish. Then check the build files into source control with the updated version number.

Whenever a new JAR version is pulled from the repository, the Studio project must be regenerated to retrieve the new dependency for the Configuration module. Regenerate the Studio project by running gwb idea on the command line. Alternatively, regenerate the project in Studio by selecting **Tools > Generate Project**. Afterward, create the Schema JAR by running gwb genSchemaJar. Finally, build the application by either running gwb compile on the command line or selecting **Build > Rebuild Project** in Studio.

Setting response serialization options, including encodings

You can customize how ClaimCenter serializes the XML in the web service response to the client.

Incoming web service requests support any valid character encodings recognized by the Java Virtual Machine. The web service client determines the encoding that it uses, not the server or its WSDL.

The most commonly customized serialization option is changing character encoding. Outgoing web service responses by default use the UTF-8 character encoding. You might want to use another character encoding for your service to improve Asian language support or other technical reasons.

To support additional serializations, add the @WsiSerializationOptions annotation to the web service implementation class. As an argument to the annotation, pass a list of XmlSerializationOptions objects. The XmlSerializationOptions class encapsulates various options for serializing XML, and that includes setting its Encoding property to a character encoding of type java.nio.charset.Charset.

The easiest way to get the appropriate character set object is to use the Charset.forName(*ENCODING_NAME*) static method. That method returns the desired static instance of the character set object.

For example, add the following the annotation immediately before the web service implementation class to specify the Big5 Chinese character encoding.

```
@WsiSerializationOptions( new() { :Encoding = Charset.forName( "Big5" ) } )
```

Adding advanced security layers to a web service

For security options beyond simple HTTP authentication and Guidewire authentication, you can use an additional set of APIs to implement additional layers of security. For example, you might want to add additional layers of encryption, digital signatures, or other types of authentication or security. From the SOAP server side, you add advanced security layers to outgoing requests by applying transformations to the data stream of the request.

Applying multiple security layers to a web service

Whenever you apply multiple layers of security to your web service, the order of substeps in your request and response transformation blocks is critical. Typically, the order of substeps in your response block reverses the order of substeps in your request block. For example, if you encrypt and then add a digital signature to the response data stream, remove the digital signature before decrypting the request data stream. If you remove a security layers from your web service, ensure the remaining layers preserve the correct order of substeps in the transformation blocks.

Transformations on data streams

When calling a web service operation, the service's data stream can be transformed. For example, a request data stream can be transformed to implement an encryption security layer.

A data stream can be transformed incrementally, a byte at a time, or all at once in its entirety. The type of transformation can sometimes determine the manner in which a stream must be transformed. For example, a digital signature authentication layer must transform an entire request data stream at one time.

Multiple types of transformations can be applied to a request data stream to add multiple security layers to a web service. The order in which the transformations are performed can be important. For example, an encryption transformation followed by a digital signature transformation produces a different request data stream than if the same transformations are performed in the opposite order.

The following example performs a simple search-and-replace operation on a request data stream. An encryption operation could follow a similar process.

```
uses gw.util.StreamUtil

class sampleRequestTransform {

    function transReplace_async() : String {
        // Create "doubled" string "XX"
        var async = API.async_doubleString("X")

        // Define Gosu block to perform transformation on input stream
        async.RequestTransform = \ inputStream -> {
            var bytes = StreamUtil.getContent(inputStream)
            var content = StreamUtil.toString(bytes)
            print("Input stream was: " + content)

            // Perform replace transformation (or encryption, as desired)
            content = content.replace("X", "Y") // "XX -> "YY"
            return StreamUtil.getStringInputStream(content)
        }

        // Call the transformation block and return the result
        return async.get()
    }
}
```

Accessing data streams

To access the data stream for a request, Gosu provides an annotation (`@WsiRequestTransform`) to inspect or transform an incoming request data stream. Gosu provides another annotation (`@WsiResponseTransform`) to inspect or transform an outgoing response data stream. Both annotations take a Gosu block that takes an input stream (`java.io.InputStream`) and returns another input stream. Gosu calls the annotation block for every request or response, depending on the type of annotation.

See also

- If the transformation operates more naturally on XML elements than on a byte stream, consider using the APIs in “Request or Response XML Structural Transformations”.

Example of data stream request and response transformations

The following example implements a request transform and a response transform to apply a simple encryption security layer to a web service.

The example applies the same incremental transformation to the request and the response data streams. The transformation processes the data streams byte by byte to change the bits in each byte from a 1 to a 0 or a 0 to a 1. The

example transformation code uses the Gosu bitwise exclusive OR (XOR) logical operator (^) to perform the bitwise changes on each byte. The XOR logical operator is a *symmetrical* operation. If you apply the XOR operation to a data stream and then apply the operation again to the transformed data stream, you obtain the original data stream. Therefore, transforming the incoming request data stream by using the XOR operation encrypts the data. Conversely, transforming the outgoing response data stream by using the XOR operation decrypts the data.

```
package gw.webservice.example

uses gw.util.StreamUtil
uses gw.xml.ws.annotation.WsiWebService
uses gw.xml.ws.annotation.WsiRequestTransform
uses gw.xml.ws.annotation.WsiResponseTransform
uses java.io.ByteArrayInputStream
uses java.io.InputStream

@WsiWebService

// Specify data stream transformations for web service requests and responses.
@WsiRequestTransform(WsiTransformTestService._xorTransform)
@WsiResponseTransform(WsiTransformTestService._xorTransform)

class WsiTransformTestService {

    // Declare a static variable to hold a Gosu block that encrypts and decrypts data streams.
    public static var _xorTransform(inputStream : InputStream) : InputStream = \ inputStream ->{

        // Get the input stream, and store it in a byte array.
        var bytes = StreamUtil.getContent(inputStream)

        // Encrypt the bits in each byte.
        for (b in bytes index idx) {
            bytes[idx] = (b ^ 17) as byte // XOR encryption with "17" as the encryption mask
        }

        return new ByteArrayInputStream(bytes)
    }

    function add(a : int, b : int) : int {
        return a + b
    }
}
```

In the preceding example, the request transformation and the response transformation use the same Gosu block for transformation logic because the block uses a symmetrical algorithm. In a typical production usage, the request transformation and the response transformation use different Gosu blocks because their transformation logic differs.

See also

- “Using WSS4J for encryption, signatures, and other security headers” on page 101

Using WSS4J for encryption, signatures, and other security headers

The following example uses the Java utility WSS4J to implement encryption, digital cryptographic signatures, and other security elements (a timestamp). This example has three parts.

Part 1 of the example that uses WSS4J

The first part of the example is a utility class called `demo.DemoCrypto` that implements an input stream encryption routine that adds a timestamp, then a digital signature, then encryption. To decrypt the input stream for a request, the utility class knows how to decrypt the input stream and then validate the digital signature.

Early in the encryption (`addCrypto`) and decryption (`removeCrypto`) methods, the code parses, or inflates, the XML request and response input streams into hierarchical DOM trees that represent the XML. The methods call the internal class method `parseDOM` to parse input streams into DOM trees.

Parsing the input streams in DOM trees is an important step. Some of the encryption information, such as timestamps and digital signatures, must be added in a particular place in the SOAP envelope. At the end of the encryption and decryption methods, the code serializes, or flattens, the DOM trees back into XML request and response input streams. The methods call the internal class method `serializeDOM` to serialize DOM trees back into input streams.

```
package gw.webservice.example

uses gw.api.util.ConfigAccess
uses java.io.ByteArrayInputStream
```

```
uses java.io.ByteArrayOutputStream
uses java.io.InputStream
uses java.util.Vector
uses java.util.Properties
uses java.lang.RuntimeException
uses javax.xml.parsers.DocumentBuilderFactory
uses javax.xml.transform.TransformerFactory
uses javax.xml.transform.dom.DOMSource
uses javax.xml.transform.stream.StreamResult
uses org.apache.ws.security.message.*
uses org.apache.ws.security.*
uses org.apache.ws.security.handler.*

// Demonstration input stream encryption and decryption functions.
// The layers of security are a timestamp, a digital signature, and encryption.
class DemoCrypto {

    // Encrypt an input stream.
    static function addCrypto(inputStream : InputStream) : InputStream {
        var crypto = getCrypto()

        // Parse the input stream into a DOM (Document Object Model) tree.
        var domEnvironment = parseDOM(inputStream)

        var securityHeader = new WSSecHeader()
        securityHeader.insertSecurityHeader(domEnvironment);

        var timeStamp = new WSSecTimestamp();
        timeStamp.setTimeToLive(600)
        domEnvironment = timeStamp.build(domEnvironment, securityHeader)

        var signer = new WSSecSignature();
        signer.setUserInfo("ws-client", "client-password")
        var parts = new Vector()
        parts.add(new WSEncryptionPart("Timestamp",
            "http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd",
            "Element"))
        parts.add(new WSEncryptionPart("Body",
            gw.xsd.w3c.soap12_envelope.Body.$QNAME.NamespaceURI, "Element"));
        signer.setParts(parts);
        domEnvironment = signer.build(domEnvironment, crypto, securityHeader);

        var encrypt = new WSSecEncrypt()
        encrypt.setUserInfo("ws-client", "client-password")
        // encryptionParts=
        // {Element}{http://www.w3.org/2000/09/xmldsig#}Signature;{Content}
        // {http://schemas.xmlsoap.org/soap/envelope/}Body
        parts = new Vector()
        parts.add(new WSEncryptionPart("Signature", "http://www.w3.org/2000/09/xmldsig#", "Element"))
        parts.add(new WSEncryptionPart("Body", gw.xsd.w3c.soap12_envelope.Body.$QNAME.NamespaceURI,
            "Content"));
        encrypt.setParts(parts)
        domEnvironment = encrypt.build(domEnvironment, crypto, securityHeader);

        // Serialize the modified DOM tree back into an input stream.
        return new ByteArrayInputStream(serializeDOM(domEnvironment.DocumentElement))
    }

    // Decrypt an input stream.
    static function removeCrypto(inputStream : InputStream) : InputStream {
        // Parse the input stream into a DOM (Document Object Model) tree.
        var envelope = parseDOM(inputStream)

        var secEngine = WSSecurityEngine.getInstance()
        var crypto = getCrypto()
        var results = secEngine.processSecurityHeader(envelope, null, \ callbacks ->{
            for (callback in callbacks) {
                if (callback typeis WSPasswordCallback) {
                    callback.Password = "client-password"
                } else {
                    throw new RuntimeException("Expected instance of WSPasswordCallback")
                }
            }
        }, crypto);

        for (result in results) {
            var eResult = result as WSSecurityEngineResult
            // Note: An encryption action does not have an associated principal.
            // Only Signature and UsernameToken actions return a principal
            if (eResult.Action != WSConstants.ENCR) {
                print(eResult.Principal.Name);
            }
        }

        // Serialize the modified DOM tree back into an input stream.
        return new ByteArrayInputStream(serializeDOM(envelope.DocumentElement))
    }

    // Private function to create a map of WSS4J cryptographic properties.
}
```

```

private static function getCrypto() : org.apache.ws.security.components.crypto.Crypto {
    var cryptoProps = new Properties()
    cryptoProps.put("org.apache.ws.security.crypto.merlin.keystore.alias", "ws-client")
    cryptoProps.put("org.apache.ws.security.crypto.merlin.keystore.password", "client-password")
    cryptoProps.put("org.apache.ws.security.crypto.merlin.keystore.type", "jks")
    cryptoProps.put("org.apache.ws.security.crypto.merlin.file", ConfigAccess.getConfigFile(
        "config/etc/client-keystore.jks").CanonicalPath)
    cryptoProps.put("org.apache.ws.security.crypto.provider",
        "org.apache.ws.security.components.crypto.Merlin")
    return org.apache.ws.security.components.crypto.CryptoFactory.getInstance(cryptoProps)
}

// Private function to parse an input stream into a hierarchical DOM tree.
private static function parseDOM(inputStream : InputStream) : org.w3c.dom.Document {
    var factory = DocumentBuilderFactory.newInstance();
    factory.setNamespaceAware(true);
    return factory.newDocumentBuilder().parse(inputStream);
}

// Private function to serialize a hierarchical DOM tree into an input stream.
private static function serializeDOM(element : org.w3c.dom.Element) : byte[] {
    var transformerFactory = TransformerFactory.newInstance();
    var transformer = transformerFactory.newTransformer();
    var baos = new ByteArrayOutputStream();
    transformer.transform(new DOMSource(element), new StreamResult(baos));
    return baos.toByteArray();
}
}

```

Part 2 of the example that uses WSS4J

The next part of this example is a web service implementation class written in Gosu. The following sample Gosu code implements the web service in the class `demo.DemoService`.

```

package gw.webservice.example

uses gw.xml.ws.annotation.WsiWebService
uses gw.xml.ws.annotation.WsiRequestTransform
uses gw.xml.ws.annotation.WsiResponseTransform
uses gw.xml.ws.annotation.WsiPermissions
uses gw.xml.ws.annotation.WsiAvailability

@WsiWebService
@WsiAvailability(NONE)
@WsiPermissions({})
@WsiRequestTransform(\ inputStream ->DemoCrypto.removeCrypto(inputStream))
@WsiResponseTransform(\ inputStream ->DemoCrypto.addCrypto(inputStream))

// This web service computes the sum of two integers. The web service decrypts incoming SOAP
// requests and encrypts outgoing SOAP responses.
class DemoService {

    // Compute the sum of two integers.
    function add(a : int, b : int) : int {
        return a + b
    }
}

```

Notice the following implementation details in this web service implementation class.

- The web service provides a method that adds two numbers, but the service itself has a request and response transformation.
- The request transformation removes and confirms the cryptographic layer on the request, including the digital signature and encryption, by calling `DemoCrypto.removeCrypto(inputStream)`.
- The response transformation adds the cryptographic layer on the response by calling `DemoCrypto.addCrypto(inputStream)`.

Part 3 of the example that uses WSS4J

The third and final part of this example is code to test this web service.

```

var webService = new wsi.local.demo.demoservice.DemoService()
webService.Config.RequestTransform = \ inputStream ->demo.DemoCrypto.addCrypto(inputStream)

```

```
webService.Config.ResponseTransform = \ inputStream ->demo.DemoCrypto.removeCrypto(inputStream)
print(webService.add(3, 5))
```

Paste this code into the Gosu Scratchpad and run it.

Locale and language support

By default, web services use the default server locale and language. Web service clients can override this behavior and set a locale and language to use while processing a web service request.

To set the locale, add a SOAP header element type `<gwsoap:gw_locale>` with the namespace "`http://guidewire.com/ws/soapheaders`". The element text specifies the desired locale code.

To set the language, add the element type `<gwsoap:gw_language>` and specify the desired language code.

The following example SOAP envelope contains a SOAP header that sets the locale and language to `fr_FR`.

```
<soap12:Envelope xmlns:soap12="http://www.w3.org/2003/05/soap-envelope">
<soap12:Header>
  <gwsoap:gw_locale xmlns:gwsoap="http://guidewire.com/ws/soapheaders">fr_FR</gwsoap:gw_locale>
  <gwsoap:gw_language xmlns:gwsoap="http://guidewire.com/ws/soapheaders">fr_FR</gwsoap:gw_language>
</soap12:Header>
<soap12:Body>
  ...
</soap12:Body>
</soap12:Envelope>
```

See also

- “Setting locale or language in a Guidewire application” on page 119

Checking for duplicate external transaction IDs

To detect duplicate operations from external systems that change data, add the `@WsCheckDuplicateExternalTransaction` annotation to your web service implementation class. To apply this feature for all operations on the service, add the annotation at the class level. To apply only to some operations, declare the annotation at the method level for individual operations.

If you apply this feature to an operation or to the whole class, ClaimCenter checks for the SOAP header `<transaction_id>` in namespace `http://guidewire.com/ws/soapheaders`. If the SOAP header `<transaction_id>` is missing, ClaimCenter throws an exception.

If the header exists, the text data is the external transaction ID that uniquely identifies the transaction. The recommended pattern for the transaction ID is to begin with an identifier for the external system, then a colon, then an ID that is unique to that external system. The most important thing is that the transaction ID be unique across all external systems.

If the web service changes any database data, the application stores the transaction ID in an internal database table for future reference. If in the future, some code calls the web service again with the same transaction ID, the database commit fails and throws the following exception.

```
com.guidewire.pl.system.exception.DBAlreadyExecutedException
```

The caller of the web service can detect this exception to identify the request as a duplicate transaction.

Because this annotation relies on database transactions (bundles), if your web service does not change any database data, this API has no effect.

See also

- If your client code is written in Gosu, to set the SOAP header see “Setting Guidewire transaction IDs on web service requests” on page 117.

Exposing typelists and enums as enumeration values and string values

For each typelist type or enumeration (Gosu or Java), the web service by default exposes this data as an enumeration value in the WSDL. This applies to both requests and responses for the service.

```
<xs:simpleType name="HouseType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="apartment"/>
    <xs:enumeration value="house"/>
    <xs:enumeration value="shack"/>
    <xs:enumeration value="mobilehome"/>
  </xs:restriction>
</xs:simpleType>
```

The published web service validates any incoming values against the set of choices and throws an exception for unknown values. Depending on the client implementation, the web service client might check if responses contain only allowed enumeration values during de-serialization. For typical cases, this approach is the desired behavior for both server and client.

For example, suppose you add new codes to a typelist or enumeration for responses. If the service returns an unexpected value in a response, it might be desirable that the client throws an exception. System integrators would quickly detect this unexpected change. The client system can explicitly refresh the WSDL and carefully check how the system explicitly handles any new codes.

However, in some cases you might want to expose enumerations as `String` values instead.

- The WSDL for a service that references typelist or enumeration types can grow in size significantly. This is true especially if some typelists or enumerations contain a vast number of choices. The number of values for a Java enumeration type cannot exceed 2000.
- Changing enumeration values even slightly can cause an incompatible WSDL. Forcing the web service client to refresh the WSDL might exacerbate upgrade issues on some projects. Although the client can refresh the WSDL from the updated service, sometimes this is an undesirable requirement. For example, perhaps new enumeration values on the server are predictably irrelevant to an older external system.
- In some cases, your actual web service client code might be middleware that passes through `String` values from another system. In such cases, you may not require explicit detection of enumeration changes in the web service client code.

To expose typelist types and enumerations (from Gosu or Java) as a `String` type, add the `@WsiExposeEnumAsString` annotation before the web service implementation class. In the annotation constructor, pass the typelist or enumeration type as the argument. For example, to expose the `HouseType` enumeration as a `String`, add the following line before the web service implementation class declaration:

```
@WsiExposeEnumAsString(HouseType)
```

To expose multiple types as `String` values, repeat the annotation for each individual type on a separate corresponding line. In this case, each separate line provides a different type as an argument. You can also add the `@WsiExposeEnumAsString` value as a default.

Optional stateful session affinity using cookies

By default, web services do not cause the application server to generate and return session cookies. However, ClaimCenter supports cookie-based load-balancing options for web services. This is sometimes referred to as *session affinity*.

Guidewire discourages session affinity with WS-I web services. Similarly, Guidewire discourages storing any state for a session in memory. Instead, read and write any state to the database for each request.

The web services layer can generate a cookie for a series of API calls. You can configure load balancing routers to send consecutive SOAP calls in the same conversation to the same server in the cluster. This feature simplifies things like

portal integration. Repeated page requests by the same user, assuming successful reuse of the cookie, go to the same node in the cluster.

By using session affinity, you can improve performance by ensuring that caches for that node in the cluster tend to already contain recently used objects and any session-specific data.

To create cookies for the session, append the text `?stateful` to the API URL.

From Gosu client code, you can use code similar to following to append text to the URL.

```
uses gw.xml.ws.WsdlConfig
uses java.net.URI
uses wsi.remote.gw.webservice.ab.a1000.abcontactapi.ABContactAPI

// Get a reference to an API object
var api = new ABContactAPI()

// Get URL and override URL to force creation of local cookies to save session for load balancing
api.Config.ServerOverrideUrl = new URI(ABContactAPI.ADDRESS + "?stateful")

// Call whatever API method you need as you normally would...
api.getReplacementAddress("12345")
```

The code might look very different depending on your choice of web service client programming language and SOAP library.

Calling web services from Gosu

Gosu code can import SOAP API web services from external systems and call these services as a SOAP client. The Gosu language handles all aspects of object serialization, object deserialization, basic authentication, and SOAP fault handling.

Gosu supports calling SOAP web services that are compliant with the WS-I Basic Profile specification. WS-I is an open industry organization that promotes industry-wide best practices for web services interoperability among diverse systems. The organization provides several different profiles and standards. The WS-I Basic Profile is the baseline for interoperable web services and more consistent, reliable, and testable APIs.

Gosu offers native WS-I web service client code with the following features.

- Call web service methods with natural Gosu syntax for method calls.
- Call web services optionally asynchronously.
- Support one-way web service methods.
- Separately encrypt requests and responses.
- Process attachments that use the multi-step MTOM protocol.
- Sign incoming responses with digital signatures.

One of the big differences between WS-I and older styles of web services is how the client and server encodes API parameters and return results. When you use the WS-I standards, you can use the encoding called Document Literal encoding (`document/literal`). The document-literal-encoded SOAP message contains a complete XML document for each method argument and return value. The schema for each of these documents is defined in a XSD file. The WSDL that describes how to talk to the published WS-I service includes a complete XSD describing the format of the embedded XML document. The outer message is very simple, and the inner XML document contains all of the complexity. Anything that an XSD can define becomes a valid payload or return value.

The WS-I standard supports a mode called RPC Literal (`RPC/literal`) instead of Document Literal. Despite the similarity in name, WS-I RPC Literal mode is not closely related to RPC encoding (RPCE). Gosu supports the WS-I RPC Literal mode for Gosu web service client code. Gosu supports RPC Literal mode by automatically converting WSDL in RPC Literal mode to equivalent WSDL in Document Literal mode.

An external web service that does not conform to the WS-I Basic Profile, such as an RPC/encoded web service or a service that uses a non-conforming callback scenario, can still be called from Gosu. To communicate with such a web service, customized code can be written using a web services engine framework, such as Apache Axis/Axis2. The framework creates Java stubs for the web service, and the generated stubs can be called like a common class method.

Loading WSDL locally by using web service collections

The recommended way of consuming WS-I web services is to use a web service collection in Guidewire Studio™. A web service collection encapsulates one or more web service endpoints and stores any WSDL or XSD files that they reference. A web service collection file includes the set of resources necessary to connect to a web service on an external server. Collection files have a .wsc file extension.

You can refresh the WSDL or XSD files in a specific web service collection from the external servers that publish web services. In Studio, navigate to a web service collection. Collections are stored in the **configuration > gsrc > wsi** hierarchy. Open a collection file to show it in the Web Service editor. To fetch the collection's resources, click the **Fetch** icon in the editor toolbar.

To specify a particular web service environment, click **File > Settings**. The Studio **Settings** windows opens. Select **Guidewire Studio** from the left Sidebar. The **Web Services** section includes an **Environment** field. Select the environment from the field's drop-down list or enter an environment name in the text box. Multiple environments can be referenced in a comma-separated list. As the collection's resources are fetched, the application resources for the configured web service environment are retrieved.

You can also refresh web service collections from the command line by running the following command:

```
gwb genFromWsc
```

Note: You cannot specify **local**: as the protocol to load a web service's WSDL. You must use the URL of the server or a variable that resolves to a valid URL, and you must ensure that the server is running before loading its WSDL.

Note: While you can specify multiple environments in the **Environment** field, the ClaimCenter server runs in only one environment. In the case of a server cluster, ClaimCenter requires that all servers in the cluster to start in the same environment.

Additionally, it is invalid to specify multiple identical environment-server pairs in a configuration. Such configuration elements are duplicates and result in runtime errors.

Loading WSDL directly into the file system for testing purposes

For testing, or in other unusual cases, you might want to load WSDL directly to the file system without using a web service collection.

Guidewire strongly recommends that you create a web service collection for consuming external web services instead of loading WSDL directly to the file system.

To consume an external web service, put the appropriate WSDL and XML schema files (XSDs) in the Gosu class hierarchy on your local system. Place them in the directory that corresponds to the package in which you want new types to appear. For example, place the WSDL and XSD files next to the Gosu class files that call the web service, organized in package hierarchies just like class files.

The following sample Gosu code shows how to manually fetch web service WSDLs for test purposes or for command-line use from a web service server.

```
uses gw.xml.ws.*
uses java.net.URL
uses java.io.File

// Set the web service endpoint URL for the web service WSDL
var urlStr = "http://www.aGreatWebService.com/GreatWebService?wsdl"

// Set the location in your file system for the web service WSDL
var loc = "/wsi/remote/GreatWebService"

// Load the web service WSDL into Gosu
Wsdl2Gosu.fetch(new URL(urlStr), new File(loc))
```

The `urlStr` variable stores the URL to the web service WSDL. The `loc` variable contains the path on your file system where the fetched WSDL is stored. You can run your version of the preceding sample Gosu code in the Gosu Scratchpad.

Types of client connections

From Gosu, there are three types of WS-I web service client connections.

- Standard round trip methods (synchronous request and response)
- Asynchronous round trip methods (send the request and immediately return to the caller, and check later to see if the request finished).
- One-way methods, which indicate a method defined to have no SOAP response at all.

How Gosu processes WSDL

Studio adds a WSDL file to the class hierarchy automatically for each registered Web Service Collection in the Web Services editor in Studio. Gosu creates all the types for your web service in the namespace `ws.web_service_name`. For this example, assume that the name of your web service is `MyService`. Gosu creates all the types for your web service in the namespace `gw.config.webservices.MyService`.

Suppose you add a Web Service in Studio and you name the web service `MyService`. Gosu creates all the types for your web service in the following namespace.

```
ws.myservice.*
```

Suppose you add a WSDL file directly to your class hierarchy called `MyService.wsdl` in the package `example.pl.gs.wsic`. Gosu creates all the types for your web service in the following namespace.

```
example.pl.gs.wsic.myservice.*
```

The name of `MyService` becomes lowercase `myservice` in the package hierarchy for the XML objects because the Gosu convention for package names is lowercase. There are other name transformations as Gosu imports types from XSDs.

The structure of a WSDL comprises the following items.

- One or more services
- For each service, one or more ports

A port represents a protocol or other context that might change how the WSDL defines that service. For example, methods might be defined differently for different versions of SOAP, or an additional method might be added for some ports. WSDL might define one port for SOAP 1.1 clients, one port for SOAP 1.2 clients, one port for HTTP non-SOAP access, and so on. See discussion later in this topic for what happens if multiple ports exist in the WSDL.

- For each port, one or more methods

A method, also called an operation or action, performs a task and optionally returns a result. The WSDL includes XML schemas (XSDs), or it imports other WSDL or XSD files. Their purposes are to describe the data for each method argument type and each method return type.

Suppose the WSDL has the following hierarchy.

```
<wsdl>
  <types>
    <schemas>
      <import schemaLocation="yourschema.xsd"/>
      <!-- Now define various operations (API methods) in the WSDL ... -->
```

The details of the web service APIs are omitted in this example WSDL. Assume the web service contains exactly one service called `SayHello`, and that service contains one method called `helloWorld`. For this first example, assume that the method takes no arguments, returns no value, and is published with no authentication or security requirements.

In Gosu, you can call the remote service represented by the WSDL.

```
// Get a reference to the web service API object in the namespace of the WSDL
// Warning: This object is not thread-safe. Do not save in a static variable or singleton instance var.

var service = new ws.myservice.SayHello()

// Call a method on the service
service.helloworld()
```

Of course, real APIs need to transfer data also. In our example WSDL, notice that the WSDL refers to a secondary schema called `yourschema.xsd`.

Studio adds any attached XSDs into the `web_service_name.wsdl.resources` subdirectory.

Let us suppose the contents of your `yourschema.xsd` file look like the following.

```
<schema>
  <element name="Root" type="xsd:int"/>
</schema>
```

Note that the element name is "root" and it contains a simple type (`int`). This XSD represents the format of an element for this web service. The web service could declare a `<root>` element as a method argument or return type.

Now let us suppose there is another method in the `SayHello` service called `doAction` and this method takes one argument that is a `<root>` element.

In Gosu, you can call the remote service represented by the WSDL.

```
// Get a Gosu reference to the web service API object
// Warning: This object is not thread-safe. Do not save in a static variable or singleton instance var.

var service = new ws.myservice.SayHello()

// Create an XML document from the WSDL by using the Gosu XML API
var x = new ws.myservice.Root()

// Call a method that the web service defines
var ret = service.doAction( x )
```

The package names are different if you place your WSDL file in a different part of the package hierarchy.

If you use Guidewire Studio, you do not need to manipulate the WSDL file manually. Studio automates getting the WSDL and saving it when you add a Web Service in the user interface.

For each web service API call, Gosu first evaluates the method parameters. Gosu serializes the root `XmlElement` instance and its child elements into XML raw data using the associated XSD data from the WSDL. Next, Gosu sends the resulting XML document to the web service. In the SOAP response, the main data is an XML document, whose schema is contained in (or referenced by) the WSDL.

Web service API objects are not thread safe

To create a WS-I API object, use a new expression to instantiate the appropriate fully-qualified type.

```
var service = new ws.myservice.SayHello()
```

Be warned that the WS-I API object is not thread-safe in all cases. It is dangerous to modify any configuration when another thread might have a connection open. Also, some APIs may directly or indirectly modify the state on the API object itself.

For example, the `initializeExternalTransactionIdForNextUse` method saves information that is specific to one request, and then resets the transaction ID after one use.

It is safest to follow the following rules.

- Instantiate the WS-I API object each time it is needed. This careful approach allows you to modify the configuration and use API without concerns for thread-safety.
- Do not save API objects in static variables.

- Do not save API objects in instance variables of classes that are singletons, such as plugin implementations. Each member field of the plugin instance becomes a singleton and needs to be shared across multiple threads.

Working with web service data by using Gosu XML APIs

All WS-I method argument types and return types are defined from schemas (the XSD embedded in the WSDL). From Gosu, all these objects are instances of subclasses of `XmlElement`, with the specific subclass defined by the schemas in the WSDL. From Gosu, working with WS-I web service data requires that you understand Gosu XML APIs.

In many cases, Gosu hides much of the complexity of XML so you do not need to worry about it. For example, for XSD-types, in Gosu you do not have to directly manipulate XML as bytes or text. Gosu handles common types like number, date, or Base64 binary data. You can directly get or set values (such as a `Date` object rather than a serialized `xsd:date` object). The `XmlElement` class, which represents an XML element hide much of the complexity.

The following items describe notable tips to working with XML in Gosu.

- When using a schema (an XSD, or a WSDL that references an XSD), Gosu exposes shortcuts for referring to child elements using the name of the element.
- When setting properties in an XML document, Gosu creates intermediate XML element nodes in the graph automatically. Use this feature to significantly improve the readability of your XML-related Gosu code.
- For properties that represent child elements that can appear more than once, Gosu exposes that property as a list. For list-based types like this, there is a special shortcut to be aware of. If you assign to the list index equal to the size of the list, then Gosu treats the index assignment as an insertion. This is also true if the size of the list is zero: use the `[0]` array/list index notation and set the property. This inserts the value into the list, which is equivalent to adding an element to the list. However, you do not have to worry about whether the list exists yet. If you are creating XML objects in Gosu, by default the lists do not yet exist.

In other words, use the following syntax.

```
element.PropertyName[0] = childElement
```

If the list does not exist yet for a list property at all, Gosu creates the list upon the first insertion. In other words, suppose an element contains child elements that represent an address and the child element has the name `Address`. If the XSD declares the element could exist more than once, the `element.Address` property is a list of addresses. The following code creates a new `Address`:

```
element.Address[0] = new my.package.xsdname.elements.Address()
```

What Gosu creates from your WSDL

Within the namespace of the package of the WSDL, Gosu creates some new types.

For each service in the web service, Gosu creates a service by name. For example, if the external service has a service called `GeocodeService` and the WSDL is in the package `examples.gosu.wsdl`, then the service has the fully-qualified type `examples.gosu.wsdl.GeocodeService`. Create a new instance of this type, and you then you can call methods on it for each method.

For each operation in the web service, generally speaking Gosu creates two local methods.

- One method with the method name in its natural form, for example, suppose a method is called `doAction`.
- One method with the method name with the `async_` prefix, for example, `async_doAction`. This version of the method handles asynchronous API calls.

Special behavior for multiple ports

Gosu automatically processes ports from the WSDL identified as either SOAP 1.1 or SOAP 1.2. If both are available for any service, Gosu ignores the SOAP 1.1 ports. In some cases, the WSDL might define more than one available port (such as two SOAP 1.2 ports with different names).

For example, suppose you add a WSDL file to your class hierarchy called `MyService.wsdl` in the package `example.pl.gs.wsic`. Gosu chooses a default port to use and creates types for the web service at the following path.

```
ROOT_PACKAGE.WSDL_NAME_NORMALIZED.NORMALIZED_SERVICE_NAME
```

The `NORMALIZED_SERVICE_NAME` name of the package is the name of the service as defined by the WSDL, with capitalization and conflict resolution as necessary. For example, if there are two services in the WSDL named `Report` and `Echo`, then the API types are in the following location.

```
example.pl.gs.wsic.myservice.Report
example.pl.gs.wsic.myservice.Echo
```

Gosu chooses a default port for each service. If there is a SOAP 1.2 version, Gosu prefers that version.

Additionally, Gosu provides the ability to explicitly choose a port. For example, if there is a SOAP 1.1 port and a SOAP 1.2 port, you could explicitly reference one of those choices. Gosu creates all the types for your web service ports within the `ports` subpackage, with types based on the name of each port in the WSDL.

```
ROOT_PACKAGE.WSDL_NAME_NORMALIZED.ports.SERVICE_AND_PORT_NAME
```

The `SERVICE_AND_PORT_NAME` is the service name, followed by an underscore, followed by the port name.

For example, suppose the ports are called `p1` and `p2` and the service is called `Report`. Gosu creates types within the following packages.

```
example.pl.gs.wsic.myservice.ports.Report_p1
example.pl.gs.wsic.myservice.ports.Report_p2
```

Additionally, if the port name happens to begin with the service name, Gosu removes the duplicate service name before constructing the Gosu type. For example, if the ports are called `ReportP1`, and `helloP2`, Gosu creates types within the following packages.

```
example.pl.gs.wsic.myservice.ports.Report_P1      // NOTE: it is not Report_ReportP1
example.pl.gs.wsic.myservice.ports.Report_helloP2 // not a duplicate, so Gosu does not remove "Hello"
```

Each one of those hierarchies would include method names for that port for that service.

Request XML complexity affects appearance of method arguments

A WS-I operation defines a request element. If the request element is simply a sequence of elements, Gosu converts these elements into multiple method arguments for the operation. For example, if the request XML has a sequence of five elements, Gosu exposes this operation as a method with five arguments.

If the request XML definition uses complex XML features into the operation definition itself, Gosu does not extract individual arguments. Instead Gosu treats the entire request XML as a single XML element based on an XSD-based type.

For example, if the WSDL defines the operation request XML with restrictions or extensions, Gosu exposes that operation in Gosu as a method with a single argument. That argument contains one XML element with a type defined from the XSD.

Use the regular Gosu XML APIs to navigate that XML document from the XSD types in the WSDL.

Adding configuration options to a web service

If a web service does not require encryption, authentication, or digital signatures, instantiate the service object and call methods on it.

```
// Get a reference to the service in the package namespace of the WSDL
var api = new example.gosu.wsi.myservice.SayHello()
```

```
// Call a method on the service
api.helloWorld()
```

If a web service requires encryption, authentication, or digital signatures, there are two approaches available.

- Configuration objects on the service instance

Set the configuration options directly on the configuration object for the service instance. The service instance is the newly-instantiated object that represents the service. In the previous example, the `api` variable holds a reference for the service instance. That object has a `Config` property that contains the configuration object of type `WsdlConfig`.

- Configuration providers in Studio

Add one or more WS-I web service configuration providers in the Studio Web Service Collection editor **Settings** tab.

Directly modifying the WSDL configuration object for a service

To add authentication or security settings to a web service you can do so by modifying the options on the service object. To access the options from the API object (in the previous example, the object in the variable called `api`), use the syntax `api.Config`. That property contains the API options object, which has the type `gw.xml.ws.WsdlConfig`.

The WSDL configuration object has properties that add or change authentication and security settings. The `WsdlConfig` object itself is not an XML object (it is not based on `XmlElement`), but some of its subobjects are defined as XML objects. Fortunately, in typical code you do not need to really think about that difference. Instead, use a straightforward syntax to set authentication and security parameters.

For XSD-generated types, if you set a property several levels down in the hierarchy, Gosu adds any intermediate XML elements if they did not already exist. This makes your XML-related code look concise.

Centralizing your WSDL configuration by adding configuration provider classes

You can add one or more WS-I web service configuration providers in the Studio Web Service Collection editor **Settings** tab. A web service configuration provider centralizes and encapsulates the steps required to add encryption, authentication, or digital signatures to your web service client code. You can also define different settings based on environment or server settings at run time.

For example, instead of adding encryption and authentication in your code each time that you call out to a single service, you can centralize that code. Using a configuration provider has the side effect of making your web service client code look cleaner. This approach separates the code that requests the web service call from the transport-layer authentication and security configuration.

A configuration provider is a class that implements the interface `gw.xml.ws.IWsServiceConfigurationProvider`. This interface defines a single method.

```
function configure( serviceName : QName, portName : QName, config : WsdlConfig )
```

The arguments are as follows.

- The service name, as a `QName`. This is the service as defined in the WSDL for this service.
- The port name, as a `QName`. Note that this is not a TCP/IP port. This is a port in the sense of the WSDL specification.
- A WSDL configuration object, which is the `WsdlConfig` object that an API service object contains each time you instantiate the service.

You can write zero, one, or more than one configuration providers and attach them to a web service collection. This means that for each new connection to one of those services, each configuration provider has an opportunity to (optionally) add configuration options to the `WsdlConfig` object.

For example, you could write one configuration provider that adds all configuration options for web services in the collection, or write multiple configuration providers that configure different kinds of options. For an example of multiple configuration providers, you could implement the following solution.

- Add one configuration provider that knows how to add authentication.
- Add a second configuration provider that knows how to add digital signatures.
- Add a third configuration provider that knows how to add an encryption layer.

Separating out these different types of configuration could be advantageous if you have some web services that share some configuration options but not others. For example, perhaps all your web service collections use digital signatures and encryption, but the authentication configuration provider class might be different for different web service collections.

The list of configuration provider classes in the Studio editor is an ordered list. For typical systems, the order is very important. For example, performing encryption and then a digital signature results in different output than adding a digital signature and then adding encryption. You can change the order in the list by clicking a row and clicking **Move Up** or **Move Down**. You can configure these settings by environment (**Env**) or server (**Server**). For default items, set them first in the list and leave the **Env** or **Server** fields blank to indicate it as a default. For example, if you have a default configuration provider, set that item to a blank **Env** setting and move it first in the list. Later items that might specify a non-blank **Env** setting will override that default value. Follow a similar approach for any defaults for any default settings using the **Server** settings.

The list of configuration providers in the Studio editor is an ordered list. If you use more than one configuration provider, carefully consider the order you want to specify them. For any defaults, such as those with blank **Env** or **Server** fields, put those at the top (beginning) of the list.

The following is an example of a configuration provider.

```
package wsi.remote.gw.webservice.ab

uses javax.xml.namespace.QName
uses gw.xml.ws.WsdlConfig
uses gw.xml.ws.IWsiWebserviceConfigurationProvider

class ABConfigurationProvider implements IWsiWebserviceConfigurationProvider {

    override function configure(serviceName : QName, portName : QName, config : WsdlConfig) {
        config.Guidewire.Authentication.Username = "su"
        config.Guidewire.Authentication.Password = "gw"
    }
}
```

In this example, the configuration provider adds Guidewire authentication to every connection that uses that configuration provider. Any web service client code for that web service collection does not need to add these options with each use of the service.

Authentication schemes for consuming WS-I web services from Gosu

When consuming web services from Gosu, there are several built-in authentication schemes. Set an authentication scheme by setting special properties on the **WsdlConfig** object that you can get from the **Config** property on the API instance.

```
// Create an API service instance
var apiService = new example.gosu.wsi.myservice.SayHelloAPI()

// Get the WsdlConfig object for that service
var c = apiService.Config

// Set authentication properties on the WsdlConfig "c" object...
```

The built-in authentication schemes are HTTP basic authentication, HTTP digest authentication, and Guidewire authentication.

When you call a method on the WS-I API service object, authentication happens as follows.

1. Gosu reviews properties on the **WsdlConfig** object in the **Config** property on the API instance.

2. The server looks for properties for Guidewire authentication. If they exist, Gosu attempts to authenticate using Guidewire authentication.
3. The server looks for properties for HTTP basic authentication. If they exist, Gosu attempts to authenticate using HTTP basic authentication.
4. The server looks for properties for HTTP digest authentication. If they exist, Gosu attempts to authenticate using HTTP basic authentication.
5. The server attempts to connect to the service without authentication.

HTTP basic authentication

HTTP basic authentication is the most common type of authentication for consuming a web service that is not published by a Guidewire application. HTTP authentication requests that the web server authenticate the request. If you are connecting to a Guidewire application, instead use Guidewire authentication, which authenticates against the user database.

User names and passwords are sent unencrypted in HTTP Basic authentication and Guidewire authentication schemes. If you use HTTP basic or Guidewire authentication across an insecure network, it is critical that you wrap the web service requests with HTTPS/SSL connections to protect security credentials. In contrast, the HTTP digest authentication uses a more secure hashing scheme that does not send the password unencrypted. However, even with HTTP digest authentication, the request and response are unencrypted. Therefore using HTTPS/SSL connections is still valuable for overall data security over insecure networks.

To add HTTP basic authentication to an API request, use the HTTP basic authentication object at the path as follows. Suppose *api* is a reference to a SOAP API that you have already instantiated with the new operator. The properties on which to set the name and password are on the object.

```
api.Config.Http.Authentication.Basic
```

That object has a **Username** property for the user name and a **Password** property for the password. Set those two values with the desired user name and password.

```
// Get a reference to the service in the package namespace of the WSDL.  
var service = new example.gosu.wsi.myservice.SayHelloAPI()  
  
service.Config.Http.Authentication.Basic.Username = "jms"  
service.Config.Http.Authentication.Basic.Password = "b5"  
  
// Call a method on the service.  
service.helloworld()
```

HTTP digest authentication

User names and passwords are sent unencrypted in HTTP digest authentication and Guidewire authentication schemes. In contrast, the HTTP digest authentication uses a more secure hashing scheme that does not send the password unencrypted.

Even with HTTP digest authentication, the request and response are unencrypted. Using HTTPS/SSL connections is still valuable for overall data security over insecure networks.

To add HTTP digest authentication to an API request, use the HTTP digest authentication object at the path as follows. Suppose *api* is a reference to a SOAP API that you have already instantiated with the new operator. The properties on which to set the name and password are on the object:

```
api.Config.Http.Authentication.Digest
```

That object has a **Username** property for the user name and a **Password** property for the password. Set those two values with the desired user name and password.

```
// Get a reference to the service in the package namespace of the WSDL.  
var service = new example.gosu.wsi.myservice.SayHelloAPI()  
  
service.Config.Http.Authentication.Digest.Username = "jms"
```

```
service.Config.Http.Authentication.Digest.Password = "b5"
// Call a method on the service.
service.helloworld()
```

Guidewire authentication

If you are connecting to a Guidewire application, use Guidewire authentication, which authenticates against the user database. To consume a web service that is not published by a Guidewire application, use Guidewire authentication.

User names and passwords are sent unencrypted in HTTP basic authentication and Guidewire authentication schemes. If you use HTTP basic or Guidewire authentication across an insecure network, it is critical that you wrap the web service requests with HTTPS/SSL connections to protect security credentials. In contrast, the HTTP digest authentication uses a more secure hashing scheme that does not send the password unencrypted. However, even with HTTP digest authentication, the request and response are unencrypted. Therefore using HTTPS/SSL connections is still valuable for overall data security over insecure networks.

To add Guidewire application authentication to API request, use the Guidewire authentication object at the path as follows. In this example, *api* is a reference to a SOAP API that you have already instantiated with the new operator:

```
api.Config.Guidewire.Authentication
```

That object has a **Username** property for the user name and a **Password** property for the password. Set those two values with the desired user name and password.

```
// Get a reference to the service in the package namespace of the WSDL.
var service = new example.gosu.wsi.myservice.SayHello()

service.Config.Guidewire.Authentication.Username = "jms"
service.Config.Guidewire.Authentication.Password = "b5"

// Call a method on the service.
service.helloworld()
```

Guidewire suite configuration file

ClaimCenter uses a single suite configuration file called `suite-config.xml` to configure web service URLs for other Guidewire InsuranceSuite applications. The suite configuration file specifies the other InsuranceSuite applications, their URLs, port numbers, and an optional environment setting, such as production or development. For InsuranceSuite integrations that use WS-I web services, always use the suite configuration file to configure InsuranceSuite web services.

To access the suite configuration file in Guidewire Studio™, go to **configuration > config > suite**, and click `suite-config.xml`. Alternatively, press **Shift+Ctrl+N**, and then enter the configuration file name.

In the base suite configuration file, all `<product>` elements are commented out. If multiple Guidewire products are on a single server for testing, the file might look like the following.

```
<suite-config xmlns="http://guidewire.com/config/suite/suite-config">
<!--<product name="cc" url="http://localhost:8080/cc"/>
<!--<product name="pc" url="http://localhost:8180/pc"/>
<!--<product name="ab" url="http://localhost:8280/ab"/>
<!--<product name="bc" url="http://localhost:8580/bc"/>
-->
```

In production, the applications would be on separate physical servers with different domain names for each application.

The optional `env` attribute can be specified in a `<product>` element to define an environment setting for the product. A single product can have multiple configurations differentiated by their environments. For example, separate ContactManager configurations can be defined for production and development environments.

```
<suite-config xmlns="http://guidewire.com/config/suite/suite-config">
<product name="ab" url="http://localhost:8080/cc"/>
<product name="ab" url="http://localhost:8080/cc" env="development"/>
<product name="ab" url="http://ProductionClaimCenterServer:8080/cc" env="production"/>
-->
```

The env attribute is optional as a technical matter. However, only one URL for a particular product can have an env attribute with a null value. A product configuration without an env attribute is interpreted as the residual default configuration. You can specify multiple system environments for the env attribute by using a comma-separated list.

When importing WS-I WSDL, ClaimCenter does the following.

1. Checks to see whether there is a WSDL port of element type <gwwsdl:address>. If such a port is found, ClaimCenter ignores any other ports on the WSDL for this service.
2. If so, it looks for shortcuts with the syntax \${productNameShortcut}. For example: \${cc}.
3. If that product name shortcut is in the suite-config.xml file, ClaimCenter substitutes the URL from the XML file to replace the text for \${productNameShortcut}. If the product name shortcut is not found, Gosu throws an exception during WSDL parsing.

For web services that Guidewire applications publish, all WSDL documents have the <gwwsdl:address> port in the WSDL. The Guidewire application automatically specifies the application that published it using the standard two-letter application shortcut. For example, for ClaimCenter the abbreviation is cc.

```
<wsdl:port name="TestServiceSoap11Port" binding="TestServiceSoap11Binding">
  <soap11:address location="http://172.24.11.41:8480/cc/ws/gw/test/TestService/soap11" />
  <gwwsdl:address location="${cc}/ws/gw/test/TestService/soap11" />
</wsdl:port>
```

Accessing the suite configuration file by using the API

ClaimCenter exposes the suite configuration file as APIs that you can access from Gosu. The gw.api.suite.GuidewireSuiteUtil class can look up entries in the file by the product code. This class has a static method called getProductId that takes a String that represents the product. Pass the String that is the <product> element's name attribute. The method returns the associated URL from the suite-config.xml file.

```
uses gw.api.suite.GuidewireSuiteUtil
var x = GuidewireSuiteUtil.getProductId("cc")
print(x.Url)
```

For the earlier suite-config.xml example file, this code prints the following output.

```
http://localhost:8080/cc
```

Setting Guidewire transaction IDs on web service requests

If the web service you are calling is hosted by a Guidewire application, there is an optional feature to detect duplicate operations from external systems that change data. The service must add the @WsiCheckDuplicateExternalTransaction annotation.

If this feature is enabled for an operation, ClaimCenter checks for the SOAP header <transaction_id> in namespace <http://guidewire.com/ws/soapheaders>.

To set this SOAP header, call the initializeExternalTransactionIdForNextUse method on the API object and pass the transaction ID as a String value.

ClaimCenter sends the transaction ID with the next method call and applies only to that one method call. If you require subsequent method calls with a transaction ID, call initializeExternalTransactionIdForNextUse again before each external API that requires a transaction ID. If your next call to an web service external operation does not require a transaction ID, there is no need for you to call the initializeExternalTransactionIdForNextUse method.

```
uses gw.xsd.guidewire.soapheaders.TransactionId
uses gw.xml.ws.WsdlConfig
uses java.util.Date
uses wsi.local.gw.services.wsidbupdateservice.faults.DBAlreadyExecutedException

function callMyWebService {

  // Get a reference to the service in the package namespace of the WSDL.
  var service = new example.gosu.wsi.myservice.SayHello()
```

```

service.Config.Guidewire.Authentication.Username = "su"
service.Config.Guidewire.Authentication.Password = "gw"

// Create a transaction ID that has an external system prefix and then a guaranteed unique ID.
// If you are using Guidewire messaging, you may want to use the Message.ID property in your ID.
transactionIDString = "MyExtSys1:" + getUniqueTransactionID() // Create your own unique ID.

// Set the transaction ID for the next method call and only the next method call to this service.
service.initializeExternalTransactionIdForNextUse(transactionIDString)

// Call a method on the service -- A transaction ID is set only for the next operation.
service.helloWorld()

// Call a method on the service -- NO transaction ID is set for this operation!
service.helloWorldMethod2()

transactionIDString = "MyExtSys1:" + getUniqueTransactionID() // Create your own unique ID.

// Call a method on the service -- A transaction ID is set only for the next operation.
service.helloWorldMethod3()
}

```

Setting a timeout on a web service

To set the timeout period in milliseconds, set the `CallTimeout` property on the `WsdlConfig` object for that API reference.

```

// Get a reference to the service in the package namespace of the WSDL
var service = new example.gosu.wsi.myservice.SayHello()

service.Config.CallTimeout = 30000 // 30 seconds

// Call a method on the service
service.helloWorld()

```

Custom SOAP headers

SOAP HTTP headers are essentially XML elements attached to the SOAP envelope for the web service request or its response. Your code might need to send additional SOAP headers to the external system, such as custom headers for authentication or digital signatures. You also might want to read additional SOAP headers on the response from the external system.

To add SOAP HTTP headers to a request that you initiate, first construct an XML element using the Gosu XML APIs (`XmlElement`). Next, add that `XmlElement` object to the list in the location `api.Config.RequestSoapHeaders`. That property contains a list of `XmlElement` objects, which in generics notation is the interface type `java.util.List<XmlElement>`.

To read (get) SOAP HTTP headers from a response, it only works if you use the asynchronous SOAP request APIs. There is no equivalent API to get just the SOAP headers on the response, but you can get the response envelope, and access the headers through that. You can access the response envelope from the result of the asynchronous API call. This is an `gw.xml.ws.AsyncResponse` object. On this object, get the `ResponseEnvelope` property. For SOAP 1.2 envelopes, the type of that response is type `gw.xsd.w3c.soap12_envelope.Envelope`. For SOAP 1.1, the type is the same except with "soap11" instead of "soap12" in the name.

From that object, get the headers in the `Header` property. That property contains a list of XML objects that represent all the headers.

See also

- “Asynchronous method calls to web services” on page 121

Server override URL

To override the server URL, for example for a test-only configuration, set the `ServerOverrideUrl` property on the `WsdlConfig` object for your API reference. This property takes a `String` object for the URL.

```

// Get a reference to the service in the package namespace of the WSDL
var service = new example.gosu.wsi.myservice.SayHello()

```

```
service.Config.ServerOverrideUrl = "http://testingserver/xx"  
// Call a method on the service  
service.helloWorld()
```

Setting XML serialization options

To send a SOAP request to the SOAP server, ClaimCenter takes an internal representation of XML and serializes the data to actual XML data as bytes. For typical use, the default XML serialization settings are sufficient. If you need to customize these settings, you can do so.

The most common serialization option to set is changing the character encoding to something other than the default, which is UTF-8.

You can change serialization settings by getting the `XmlSerializationOptions` property on the `WsdlConfig` object, which has type `gw.xml.XmlSerializationOptions`. Modify properties on that object to set various serialization settings.

The easiest way to get the appropriate character set object for the encoding is to use the `Charset.forName(ENCODING_NAME)` static method. That method returns the desired static instance of the character set object.

The following code sample changes the encoding to the Chinese encoding Big5.

```
uses java.nio.charset.Charset  
  
// Get a reference to the service in the package namespace of the WSDL  
var service = new example.gosu.wsdl.myservice.SayHello()  
  
service.Config.XmlSerializationOptions.Encoding = Charset.forName("Big5")  
  
// Call a method on the service  
service.helloWorld()
```

This API sets the encoding on the outgoing request only. The SOAP server is not obligated to return the response XML in the same character encoding.

If the web service is published from a Guidewire product, you can configure the character encoding for the response.

Setting locale or language in a Guidewire application

By default, a web service uses the locale and language configured on the server. A web service client can override these attributes and set a locale and language to use while processing a request.

```
// Get a reference to the web service in the package namespace of the WSDL  
var service = new example.gosu.wsdl.myservice.SayBonjour()  
  
service.Config.Guidewire.GwLocale = "fr_FR"      // Set locale region to France  
service.Config.Guidewire.GwLanguage = "Fr_FR"     // Set language to French  
  
// Call a method on the service  
service.BonjourToutLeMonde()
```

Implementing advanced web service security with WSS4J

For security options beyond HTTP Basic authentication and optional SOAP header authentication, you can use an additional set of APIs to implement whatever additional security layers.

For example, you might want to add additional layers of encryption, digital signatures, or other types of authentication or security.

From the SOAP client side, the way to add advanced security layers to outgoing requests is to apply transformations of the stream of data for the request. You can transform the data stream incrementally as you process bytes in the stream. For example, you might implement encryption this way. Alternatively, some transformations might require getting all the bytes in the stream before you can begin to output any transformed bytes. Digital signatures would be an example of this approach. You may use multiple types of transformations. Remember that the order of them is important.

For example, an encryption layer followed by a digital signature is a different output stream of bytes than applying the digital signature and then the encryption.

Similarly, getting a response from a SOAP client request might require transformations to understand the response. If the external system added a digital signature and then encrypted the XML response, you need to first decrypt the response, and then validate the digital signature with your keystore.

The standard approach for implementing these additional security layers is the Java utility WSS4J, but you can use other utilities as needed. The WSS4J utility includes support for the WSS security standard.

Outbound security for a web service request

To add a transformation to your outgoing request, set the `RequestTransform` property on the `WsdlConfig` object for your API reference. The value of this property is a Gosu block that takes an input stream (`InputStream`) as an argument and returns another input stream. Your block can do anything it needs to do to transform the data.

Similarly, to transform the response, set the `ResponseTransform` property on the `WsdlConfig` object for your API reference.

The following simple example shows you could implement a transform of the byte stream. The transform is in both outgoing request and the incoming response. In this example, the transform is an XOR (exclusive OR) transformation on each byte. In this simple example, simply running the transformation again decodes the request.

The following code implements a service that applies the transform to any input stream. The code that actually implements the transform is as follows. This is a web service that you can use to test this request.

The class defines a static variable that contains a field called `_xorTransform` that does the transformation.

```
package gw.xml.ws

uses gw.xml.ws.annotation.WsiWebService
uses gw.xml.ws.annotation.WsiRequestTransform
uses java.io.ByteArrayInputStream
uses gw.util.StreamUtil
uses gw.xml.ws.annotation.WsiResponseTransform
uses gw.xml.ws.annotation.WsiAvailability
uses gw.xml.ws.annotation.WsiPermissions
uses java.io.InputStream

@WsiWebService
@WsiAvailability( NONE )
@WsiPermissions( {} )
@WsiRequestTransform( WsiTransformTestService._xorTransform )
@WsiResponseTransform( WsiTransformTestService._xorTransform )
class WsiTransformTestService {

    // The following method declares a Gosu block that implements the transform
    public static var _xorTransform( is : InputStream ) : InputStream = \ is ->{
        var bytes = StreamUtil.getContent( is )
        for ( b in bytes index idx ) {
            bytes[ idx ] = ( b ^ 17 ) as byte // xor encryption
        }
        return new ByteArrayInputStream( bytes )
    }

    function add( a : int, b : int ) : int {
        return a + b
    }
}
```

The following code connects to the web service and applies this transform on outgoing requests and the reply.

```
package gw.xml.ws

uses gw.testng.TestBase
uses gw.testng.RunLevel
uses org.xml.sax.SAXParseException

@RunLevel( NONE )
class WsiTransformTest extends TestBase {

    function testTransform() {
        var ws = new wsi.local.gw.xml.ws.wsitransformtestservice.WsiTransformTestService()
        ws.Config.RequestTransform = WsiTransformTestService._xorTransform
        ws.Config.ResponseTransform = WsiTransformTestService._xorTransform
        ws.add( 3, 5 )
    }
}
```

```
}
```

One-way methods

A typical WS-I method invocation has two parts: the SOAP request, and the SOAP response. Additionally, WS-I supports a concept called *one-way methods*. A one-way method is a method defined in the WSDL to provide no SOAP response at all. The transport layer (HTTP) may send a response back to the client, however, but it contains no SOAP response.

Gosu fully supports calling one-way methods. Your web service client code does not have to do anything special to handle one-way methods. Gosu handles them automatically if the WSDL specifies a method this way.

Be careful not to confuse one-way methods with asynchronous methods.

Asynchronous method calls to web services

Gosu supports optional asynchronous calls to web services by exposing web service methods signatures with the `async_` prefix. Gosu does not generate the additional method signature if the method is a one-way method. The asynchronous method variants return an `AsyncResponse` object. Use that object with a polling design pattern by checking regularly whether it is done, then get results later, synchronously in relation to the calling code.

The `AsyncResponse` object contains the following properties and methods.

start

This method initiates the web service request but does not wait for the response.

get method

This method gets the results of the web service, waiting (blocking) until complete if necessary.

RequestEnvelope

A read-only property that contains the request XML.

ResponseEnvelope

A read-only property that contains the response XML, if the web service responded.

RequestTransform

A block (an in-line function) that Gosu calls to transform the request into another form. For example, this block might add encryption and then add a digital signature.

ResponseTransform

A block (an in-line function) that Gosu calls to transform the response into another form. For example, this block might validate a digital signature and then decrypt the data.

The following code is an example of calling the asynchronous version of a method in contrast to calling the synchronous variant.

```
var ws = new ws.weather.Weather()

// Call the REGULAR version of the method.
var r = ws.GetCityWeatherByZIP("94114")
print( "The weather is " + r.Description )

// Call the **asynchronous** version of the same method
// -- Note the "async_" prefix to the method
var a = ws.async_GetCityWeatherByZIP("94114")

// By default, the async request does NOT start automatically.
// You must start it with the start() method.
a.start()

print("The XML of the request for debugging... " + a.RequestEnvelope)
print("")
print ("In a real program, you would check the result possibly MUCH later...")

// Get the result data of this asynchronous call, waiting if necessary.
var laterResult = a.get()
```

```
print("asynchronous reply to our earlier request = " + laterResult.Description)
print("response XML = " + a.ResponseEnvelope.asUTFString())
```

MTOM attachments with Gosu as web service client

The W3C Message Transmission Optimization Mechanism (MTOM) is a method of efficiently sending binary data to and from web services as attachments outside the normal response body.

The main response contains placeholder references for the attachments. The entire SOAP message envelope for MTOM contains multiple parts. The raw binary data is in other parts of the request than the normal SOAP request or response. Other parts can have different MIME encoding. For example, it could use the MIME encoding for the separate binary data. This allows more efficient transfer of large binary data.

When Gosu is the web service client, MTOM is not supported in the initial request. However, if an external web service uses MTOM in its response, Gosu automatically receives the attachment data. There is no additional step that you need to perform to support MTOM attachments.

chapter 6

General web services

This topic describes general-purpose web services, such as mapping typecodes general system tools.

Importing administrative data

ClaimCenter provides tools for exporting and importing administrative data in XML format. The easiest way to export data suitable for import is to use the **Export Data** screen in the application. Then, you can use the Import Tools web service (`ImportToolsAPI`) to import the exported data to a different application instance.

Prepare exported administrative data for import

To prepare exported data for XML import, use the generated *XML Schema Definition* (XSD) files that define the XML data formats.

```
ClaimCenter/build/xsd/cc_import.xsd  
ClaimCenter/build/xsd/cc_entities.xsd  
ClaimCenter/build/xsd/cc_typelists.xsd
```

Regenerate the XSD files by running the build tool with the following option.

```
gwb genImportAdminDataXsd
```

Importing prepared administrative data

Administrative database tables can be imported from an XML file by calling the `importXmlData` method of the `ImportToolsAPI` web service. The `importXmlData` method does not perform extensive data validation tests on the imported data. Accordingly, the method must not be used to import data other than administrative data. The imported objects do not generate events during the import operation. ClaimCenter does not throw concurrent data change exceptions if the imported records overwrite existing records in the database.

```
importXmlData(xmlData : String) : ImportResults
```

The `xmlData` argument contains the XML data to import. The argument cannot be null or contain an empty string.

The method returns an `ImportResults` object that contains information about the import operation, such as the number of entities imported grouped by type. If an error occurred during import, the object's `isOk` method returns `false`. Descriptions of the errors can be retrieved by calling the object's `getErrorLog` method which returns an array of `String` objects.

The performance of the import operation can be improved by wrapping the XML data in CDATA tags.

```
<pre>
<![CDATA[
  <SampleElement>Sample data</SampleElement>
]]>
</pre>
```

CSV import and conversion

There are several other methods in the `ImportToolsAPI` web service to import and convert CSV (comma-separated value) data. For example, import data from simple sample data files and convert them to a format suitable for XML import, or import them directly. See the Javadoc in the implementation file for more information about the CSV methods.

- `importCsvData` – import CSV data
- `csvToXml` – convert CSV data to XML data
- `xmlToCsv` – convert XML data to CSV data

Advanced import or export

If you must import data in other formats or export administrative data programmatically, write a new web service to export administrative information one record at a time. For both import and export, if you write your own web service, be careful never to pass too much data across the network in any API call. If you send too much data, memory errors can occur. Do not try to import or export all administrative data in a dataset at once.

Maintenance tools web service

The `MaintenanceToolsAPI` web service provides a set of tools that are available if the system is at the `maintenance` run level or higher.

Using web service methods with batch processes

You can use methods of the `MaintenanceToolsAPI` web service to work with a batch process or a writer for a work queue. If you request that a batch process start or terminate, the API notifies the caller that the request has been received. You must poll the server later to see if the process failed or completed successfully.

For server clusters, a batch process runs only on servers with the `batch` server role. However, you can make the API request to any of the servers in the cluster. If the receiving server does not have the `batch` server role, the request automatically forwards to an appropriate server.

Note: For the batch process methods of the `MaintenanceToolsAPI` web service, the batch processes apply only to ClaimCenter, not additional Guidewire applications that you integrated with ClaimCenter. In particular, batch process methods called on a ClaimCenter server apply only to ClaimCenter servers, not to ContactManager servers.

Starting a batch process

You can call `MaintenanceToolsAPI` methods to start a batch process.

Note: Whenever you use the `MaintenanceToolsAPI` web service to run a batch process provided in the base configuration, identify the batch process by its code as listed in the documentation. To run a custom batch process, identify the process by the `BatchProcessType` typecode that you added for it. To be able to start your custom batch process with the `MaintenanceToolsAPI` web service, the typecode must include the `APIRunnable` category.

The following method starts a batch process by process name and returns its process ID. If the batch process is already running on the server, the method throws an exception.

```
startBatchProcess(processName : String) : ProcessID
```

The following code statement starts a batch process by process name and passes arguments to it. The method returns the process ID of the started batch process. If the batch process is already running on the server, the method throws an exception.

```
startBatchProcessWithArguments(processName : String, arguments : String[]) : ProcessID
```

Getting and validating batch process names

You can call `MaintenanceToolsAPI` methods to get valid batch process names.

The following method returns the set of valid batch process names.

```
getValidBatchProcessNames() : String[]
```

The following method determines if a batch process name is valid.

```
isBatchProcessNameValid(processName : String) : boolean
```

Checking the status of a batch process

You can call `MaintenanceToolsAPI` methods to check the status of a batch process.

The following method gets the status of the given batch process by name, indicating whether or not the process is running and, if so, its current progress. Status is returned in a `gw.api.webservice.maintenanceTools.ProcessStatus` object.

```
batchProcessStatusByName(processName : String) : ProcessStatus
```

The following method gets the status of a particular batch process invocation. Status is returned in a `gw.api.webservice.maintenanceTools.ProcessStatus` object.

- If the invocation is still running, the returned object indicates that the batch process is starting or executing, and only the `startDate` and `opsCompleted` fields are filled in.
- If the invocation has completed, the returned object contains information about the completed run.

```
batchProcessStatusByID(pid : ProcessID) : ProcessStatus
```

Note: For work queues, the status methods return the status only of the writer thread. The status methods do not check the work queue table for remaining work items. The status of a writer reports as completed after the writer finishes adding work items for a batch to the work queue. Meanwhile, many work items in the batch might remain unprocessed.

Terminating a batch process

You can call `MaintenanceToolsAPI` methods to request termination of a batch process.

The following method requests termination of a batch process by name if it is currently running. The method does not wait for the batch process to actually terminate. The method returns `true` if the request was successful or `false` if the process could not be terminated.

```
requestTerminationOfBatchProcessByName(processName : String) : boolean
```

The following method requests termination of a batch process by process ID if it is currently running. It is possible that this particular invocation could have finished and another invocation of the same batch process could have begun, in which case the method cannot request the termination of the current invocation. This method does not wait for the batch process to actually terminate. The method returns `true` if the request was successful or `false` if the process could not be terminated.

```
requestTerminationOfBatchProcessByID(pid : ProcessID) : boolean
```

See also

- “maintenance_tools command options” in the *Administration Guide*

Using web service methods with work queues

A work queue represents a pool of work items that can be processed in a distributed way across multiple threads or even multiple servers. Several web service methods query or modify the existing work queue configuration. For example, methods can get the number of worker threads configured for this server (`instances`) and the configured delay between processing each work item (`throttleInterval`).

The following code example wakes up all workers for the specified work queue across the entire cluster:

```
import gw.webservice.cc.MaintenanceToolsAPI;
import gw.api.webservice.maintenanceTools.WorkQueueConfig;
MaintenanceToolsAPI maintenanceTools = new MaintenanceToolsAPI();
//Wakes up the workers for the work queue, ActivityEsc.
maintenanceTools.notifyQueueWorkers("ActivityEsc");
```

The following code statement gets the work queue names for this instance of ClaimCenter:

```
String[] stringArray = maintenanceTools.getWorkQueueNames();
```

Use the following method to stop the specified work queue:

```
stopWorkQueueWorkers(queueName : String)
```

Use the following method to start the specified work queue:

```
startWorkQueueWorkers(queueName : String)
```

Use the following method to retrieve the status of active executors for a work queue. Each executor contains information about last 25 workers run by each executor.

```
getWQueueStatus(queueName : String) : WQueueStatus
```

Use the following method to retrieve the number of active work items for a work queue:

```
getNumActiveWorkItems(queueName : String) : int
```

Use the following method to wait on the active work items for a queue. In the base configuration, the maximum number of seconds to wait is 60 and the amount of time to sleep is 200 milliseconds. The method returns `true` if the queue is empty.

```
waitOnActiveWorkItems(queueName : String) : boolean
```

The following code statements get the number of instances and the throttle interval for the specified work queue.

```
//Obtains the work queue, ActivityEsc, and assigns the object to a temporary work queue configuration
//object, wqConfig.
WorkQueueConfig wqConfig = maintenanceTools.getWorkQueueConfig("ActivityEsc");

//Obtains the number of instances and the throttle interval for wqConfig.
Integer numInstances = wqConfig.getInstances();
Long throttleInterval = wqConfig.getThrottleInterval();
```

The following code block sets the number of instances, the throttle interval, the batch size, and the maximum poll interval before calling `setWorkQueueConfig`. Note that you must set all fields on a temporary work queue configuration object before setting the configuration for the work queue.

```
//Sets all fields on the temporary work queue configuration object, wqConfig, before setting the
//configuration for the work queue.
wqConfig.setInstances(1);
wqConfig.setThrottleInterval(999);
wqConfig.setBatchSize(20);
```

```
wqConfig.setMaxPollInterval(60000);  
//Sets the configuration values of the work queue, ActivityEsq, to the values established for the temporary  
//work queue configuration object. This method invocation only sets the work queue configuration for the  
//server that accepts the method call.  
maintenanceTools.setWorkQueueConfig("ActivityEsc", wqConfig);
```

Worker instances that are running stop after they complete their current work item. Then, the server creates and starts new worker instances as specified by the configuration object that you pass to the method.

The changes made by using the maintenance tools web service methods are temporary. If the server starts or restarts at a later time, the server rereads the values from `work-queue.xml` to define how to create and start workers.

For these APIs, the terms *product* and *cluster* apply to the current Guidewire product only as determined by the SOAP API server requested.

If you use ContactManager and you use these APIs on a ClaimCenter server, it applies only to ClaimCenter not ContactManager. Similarly, if you use these APIs on a ContactManager server, it applies only to ContactManager not ClaimCenter.

See also

- “maintenance_tools command options” in the *Administration Guide*

Using web service methods with startable plugins

A startable plugin begins executing at server startup and runs in the application server as a background process. You can call methods of the `MaintenanceToolsAPI` web service to work with startable plugins.

The startable plugins provided in the base configuration run only on servers with the `batch` server role. You can develop distributed startable plugins that run on all servers. Before you use the `MaintenanceToolsAPI` web service to stop or restart a distributed startable plugin, be certain that you understand the implications for state management across all servers.

Note: For methods that take a plugin name as a parameter, use the plugin name defined in the Plugin Registry in Guidewire Studio.

Stopping a startable plugin

You can stop a startable plugin by calling one of two `MaintenanceToolsAPI` methods.

The following method takes the plugin name as a parameter and blocks for up to 120 seconds to wait for confirmation.

```
stopPlugin(pluginName : String)
```

The following method takes the plugin name and a `timeout` in milliseconds as parameters. The method blocks for the specified timeout interval for confirmation that the plugin started up. If the value of `timeout` is -1, the method does not wait for confirmation and returns immediately.

```
stopPluginWithTimeout(pluginName : String, timeout : long)
```

Starting a startable plugin

You can determine if a plugin has started and restart a startable plugin by calling `MaintenanceToolsAPI` methods.

To determine if a startable plugin has started, call the following method. The method returns `true` if the plugin has started or `false` if the plugin has not started.

```
isPluginStarted(pluginName : String) : boolean
```

To start a startable plugin and wait for 120 seconds for confirmation of the start, call the following method:

```
startPlugin(pluginName : String)
```

To start a startable plugin and wait for a specified interval, or not wait at all, call the following method. The method takes the plugin name and a `timeout` in milliseconds as parameters. The method blocks for the specified timeout

interval for confirmation that the plugin started up. If the value of `timeout` is -1, the method does not wait for confirmation and returns immediately.

```
startPluginWithTimeout(pluginName : String, timeout : long)
```

See also

- “`maintenance_tools` command options” in the *Administration Guide*

Using maintenance tools web services to archive or restore claims

To mark claims for archive, you can call the `MaintenanceToolsAPI` web service method `scheduleForArchive`. Internally, this method calls the `scheduleForArchive` method on the `ClaimAPI` web service. It is more likely that you will use the `ClaimAPI` web service for this purpose.

Note: This method is provided on the `MaintenanceToolsAPI` web service as a convenience for the option - `scheduleforarchive` of the `maintenance_tools` command.

To restore archived claims, you can call the following `MaintenanceToolsAPI` web service method:

```
restore(claimNumbers : String[], comment : String) : String
```

The method takes a `String` array of claim numbers and a comment as the parameters and returns a `String` indicating the claims processed and skipped. This method has the same functionality as the `restoreClaims` method on the `ClaimAPI` web service.

See also

- “Archiving claims from external systems” on page 154
- “Restoring archived claims from external systems” on page 154

Marking claims for purging by using web services

You can mark claims for purging by using the `MaintenanceToolsAPI` method `markPurgeReady`, which sets the purge ready flag on the claim so the Bulk Purge batch process can purge it. The method takes as arguments a `String` array of claim numbers and a Boolean value indicating whether or not to purge the claim even if it is part of an aggregate limit. If this second parameter is false, any specified claim that is part of an aggregate limit causes the method to return an error message. The `String` return value indicates if the claims were successfully marked.

```
markPurgeReady(claimNumbers : String[], purgeFromAggregateLimit : boolean) : String
```

See also

- *Administration Guide*

Changing a Contact subtype by using web services

You can change the subtype of a `Contact` instance by using the following `MaintenanceToolsAPI` method, which has parameters for the `PublicID` of the contact and the target `Contact` subtype specified as a `String`.

Note: The value of `PublicID` is not necessarily the same for a contact stored in `ContactManager` and the equivalent contact instance or instances stored in `ClaimCenter`. Additionally, the value of `publicID` is not necessarily the same as that of the `LinkID` in `ContactManager` or the `AddressBookUID` in `ClaimCenter`.

```
changeSubtype(publicID : String, targetType : String)
```

Note: Do not use this method without first understanding the restrictions on changing the subtype of a `Contact` instance. For example, you must consider:

- Which applications store the contact
- Restrictions on client contact subtype

- Differences in fields supported by various contact subtypes

See also

- “Restrictions on changing the subtype of a Contact instance” in the *Contact Management Guide*
- “ContactManager link IDs and comparison to other IDs” in the *Contact Management Guide*
- “Troubleshooting the change subtype command-prompt utility” in the *Contact Management Guide*

Recalculating aggregate limits by using web services

You can force a recalculation of aggregate limits stored in the ClaimCenter database by calling one of three `MaintenanceToolsAPI` methods, depending on the aggregate limits you need to recalculate. The work queue then repopulates the database tables with this updated data.

An *aggregate limit* is the maximum financial amount that an insurer must pay on a policy or coverage during a given policy period.

Note: Recalculate aggregate limits only if you encounter consistency check failures and cannot identify the reason for the inconsistency.

All the following methods return a String indicating success or error. Additionally, the methods can throw `SOAPException` if any aggregate limits cannot be rebuilt or marked invalid, or if the work items cannot be created.

- Use the following method to schedule recalculation of invalid aggregate limits:

```
scheduleAggLimitRebuildInvalidLimits(): String
```

- Use the following method to mark all aggregate limits as invalid and force recalculation of all of them:

```
scheduleAggLimitRebuildAllLimits(): String
```

- Use the following method to recalculate aggregate limits for the specified claims:

```
scheduleAggLimitRebuildOfClaims(claimNumbers : String[]) : String
```

- Use the following method to recalculate aggregate limits for the specified policies:

```
scheduleAggLimitRebuildOfPolicies(policyNumbers : String[]) : String
```

See also

- *Administration Guide*

Getting the calculation date of team statistics by using web services

You can get the calculation date of the current Team page statistics by using a `MaintenanceToolsAPI` method. You might want to call this method or use the equivalent maintenance tools -whenstats command option if you want to make sure the statistics are current. The method to call is:

```
whenStatsCalculated() : Date
```

See also

- *Administration Guide*
- *ClaimCenter Application Guide*

Mapping typecodes and external system codes

In the simplest case, the typecodes used by ClaimCenter typelists and the equivalent codes used by an external system match exactly. When the typecodes between the two systems match, there is no need to map one to the other. However, this situation is rarely possible. In most cases, the codes used in one system must be mapped to the equivalent codes in the other system.

Typecode mappings are defined in an XML file called `typecodemapping.xml`. This file is located in the `ClaimCenter/modules/configuration/config/typelists/mapping` directory. This file defines mappings for one or more external

systems. Each external system is identified by a unique namespace. The example mapping file shown below maps a single ClaimCenter typelist typecode of `LossType.PR` to its equivalent code in two external systems, `ExtSys123` and `ExtSysABC`.

```
<?xml version="1.0"?>
<typecodemapping>
  <namespacelist>
    <namespace name="ExtSys123"/>
    <namespace name="ExtSysABC"/>
  </namespacelist>

  <typelist name="LossType">
    <mapping typecode="PR" namespace="ExtSys123" alias="Prop"/>
    <mapping typecode="PR" namespace="ExtSysABC" alias="CPL"/>
  </typelist>
</typecodemapping>
```

As the example file demonstrates, each external system is assigned a unique namespace name. The name is subsequently used to map a ClaimCenter typecode to the relevant external system code. The mapped ClaimCenter typecode is specified in the `typecode` attribute, and the external system code is specified in the `alias` attribute.

The `typecodemapping.xml` file can define multiple `typelist` elements. The value of each `name` attribute must be unique and cannot be an empty string.

A single ClaimCenter typecode can be mapped to multiple external system aliases. Conversely, a single external alias can be mapped to multiple typecodes. To determine the appropriate mapped typecode or alias in a particular situation, an external system can call the `TypelistToolsAPI` web service to retrieve an array of mapped typecodes or aliases and select the desired item from the array. ClaimCenter configuration code can resolve the issue in a similar manner by using the static `TypecodeMapper` object.

The `TypelistToolsAPI` web service

The `TypelistToolsAPI` web service enables an external system to translate the mappings between its codes and ClaimCenter typecodes.

The `getTypelistValues` method

The `getTypelistValues` method returns an array of the typekeys for a specified typelist.

```
getTypelistValues(typelist : String) : TypeKeyData[]
```

The `typelist` argument specifies a ClaimCenter typelist. If the typelist does not exist, the method throws an `IllegalArgumentException`. The method returns an array of `TypeKeyData` objects that contain the typekeys defined by the typelist. The typecode for a particular `TypeKeyData` object can be retrieved by calling its `getCode` method.

The `getTypeKeyByAlias` and `getTypeKeysByAlias` methods

The `getTypeKeyByAlias` and `getTypeKeysByAlias` methods are used when an external system is exporting data to ClaimCenter. The methods enable the external system to translate one of its own codes to a mapped ClaimCenter typecode.

```
getTypeKeyByAlias(typelist : String, namespace : String, alias : String ) : TypeKeyData
getTypeKeysByAlias(typelist : String, namespace : String, alias : String) : TypeKeyData[]
```

Each argument references an attribute value assigned in the `typecodemapping.xml` file. The `typelist` argument references a ClaimCenter typelist by specifying the value of the `name` attribute of the `typelist` element. The `namespace` argument identifies the external system by specifying its unique `name` attribute value assigned in its `namespace` element. The `alias` argument references the external system's code by specifying the `alias` attribute value of the `mapping` element. None of the arguments can be `null`.

The methods return either a single `TypeKeyData` object or an array of `TypeKeyData` objects that are mapped to the specified external system `alias`. The typecode for a particular `TypeKeyData` object can be retrieved by calling its `getCode` method.

If `getTypeKeyByAlias` finds more than one typecode mapped to the specified code, it throws an `IllegalArgumentException`.

The `getAliasByInternalCode` and `getAliasesByInternalCode` methods

The `getAliasByInternalCode` and `getAliasesByInternalCode` methods are used when an external system is importing data from ClaimCenter. The term "Alias" in the method names refers to an external system's code, and "InternalCode" refers to a ClaimCenter typelist and typecode combination. The methods enable the external system to translate a ClaimCenter typelist/typecode to a mapped code in the external system.

```
getAliasByInternalCode(typelist : String, namespace : String, code : String) : String  
getAliasesByInternalCode(typelist : String, namespace : String, code : String) : String[]
```

Each argument references an attribute value assigned in the `typecodemapping.xml` file. The `typelist` argument references a ClaimCenter typelist by specifying the value of the `name` attribute of the `typelist` element. The `code` argument references the typelist's typecode by specifying the `typecode` attribute value of the `mapping` element. The `namespace` argument identifies the external system by specifying its unique `name` attribute value assigned in its `namespace` element. None of the arguments can be null.

The methods return either a single external system code or an array of codes that are mapped to the specified `typelist` and `code`.

If `getAliasByInternalCode` finds more than one code mapped to the specified typelist/typecode, it throws an `IllegalArgumentException`.

The TypecodeMapper class

The `TypecodeMapper` class enables configuration code to translate the mappings between ClaimCenter typecodes and the codes used by an external system.

The `getTypecodeMapper` method

The `getTypecodeMapper` method retrieves the static `TypecodeMapper` object which is used to translate the mappings of typecodes and external system codes. The method is implemented in the `gw.api.util.TypecodeMapperUtil` class.

```
gw.api.util.TypecodeMapperUtil.getTypecodeMapper() : TypecodeMapper
```

The method accepts no arguments. It returns a `TypecodeMapper` object which implements the remaining methods described in this section.

The `containsMappingFor` method

The `containsMappingFor` method determines whether a specified typelist has any mappings to external system codes.

```
containsMappingFor(typelist : String) : boolean
```

The `typelist` argument references a ClaimCenter typelist by specifying the value of the `name` attribute of the `typelist` element defined in the `typecodemapping.xml` file. The specified typelist must exist in ClaimCenter or the method throws an `InvalidMappingFileNotFoundException`.

The method returns `true` if any mappings are defined for the specified typelist.

The `getAliasByInternalCode` and `getAliasesByInternalCode` methods

The `getAliasByInternalCode` and `getAliasesByInternalCode` methods are called when ClaimCenter is exporting data to an external system. The term "Alias" in the method names refers to an external system's code, and "InternalCode" refers to a ClaimCenter typelist and typecode combination. The methods enable the configuration code to translate a ClaimCenter typelist/typecode to a mapped code in the external system.

```
getAliasByInternalCode(typelist : String, namespace : String, code : String) : String  
getAliasesByInternalCode(typelist : String, namespace : String, code : String) : String
```

Each argument references an attribute value assigned in the `typecodemapping.xml` file. The `typelist` argument references a ClaimCenter typelist by specifying the value of the `name` attribute of the `typelist` element. The `code` argument references the typelist's typecode by specifying the `typecode` attribute value of the `mapping` element. The `namespace` argument identifies the external system by specifying its unique `name` attribute value assigned in its `namespace` element. None of the arguments can be `null`.

The methods return either a single external system code or an array of codes that are mapped to the specified `typelist` and `code`. If no mapped code is found, `getAliasByInternalCode` returns `null` and `getAliasesByInternalCode` returns an empty array.

If `getAliasByInternalCode` finds more than one code mapped to the specified `typelist/typecode`, it throws a `NonUniqueAliasException`.

[The `getInternalCodeByAlias` and `getInternalCodesByAlias` methods](#)

The `getInternalCodeByAlias` and `getInternalCodesByAlias` methods are called when ClaimCenter is importing data from an external system. The methods enable ClaimCenter to translate an external system's code to a mapped typecode.

```
getInternalCodeByAlias(typelist : String, namespace : String, alias : String) : String
getInternalCodesByAlias(typelist : String, namespace : String, alias : String) : String[]
```

Each argument references an attribute value assigned in the `typecodemapping.xml` file. The `typelist` argument references a ClaimCenter typelist by specifying the value of the `name` attribute of the `typelist` element. The `namespace` argument identifies the external system by specifying its unique `name` attribute value assigned in its `namespace` element. The `alias` argument references the external system's code by specifying the `alias` attribute value of the `mapping` element. None of the arguments can be `null`.

The methods return either a single typecode or an array of typecodes that are mapped to the specified external system `alias`. If no mapped typecode is found, `getInternalCodeByAlias` returns `null` and `getInternalCodesByAlias` returns an empty array.

If `getInternalCodeByAlias` finds more than one typecode mapped to the specified external system code, it throws a `NonUniqueTypecodeException`.

Data destruction web service

ClaimCenter provides a web service that enables external software programs to initiate and track requests to destroy data. In the base configuration, this web service, `PersonalDataDestructionAPI`, enables an external application to request:

- Destruction of a contact's data by `AddressBookUID` or by `PublicID`.
- Destruction of a user by user name.

In the base configuration, the `PersonalDataDestroyer` obtained by the class that implements `PersonalDataDestructionPlugin` uses a Contact `PublicID`. You can configure the `PublicID` to correspond to an entity other than Contact.

The requests for removal return a unique `requesterID` that can be used to check the status of the request. Additionally, this `requesterID` is available in the plugin notification call when the request has been completed.

Note: The `PersonalDataDestructionAPI` web service checks for both retired and active contacts when a contact data destruction request is received. When implementing data obfuscation for contacts, you must evaluate the need to look for related objects that are retired in the database. Retired objects can be obfuscated in the same fashion as active objects, but must be retrieved from the database with a query that is specified to look for retired objects. For example: `query.withFindRetired(true)`.

IMPORTANT: The external software program that calls the web service must request and verify destruction of data for a contact in ClaimCenter before requesting that `ContactManager` destroy the contact. If you have more than one core application installed, the external application must request that the contact be destroyed in all of

them before sending the request to ContactManager. When ContactManager processes a request to destroy a contact, it first verifies with all installed core applications that it can destroy the contact. If any core application indicates that the contact cannot be destroyed, ContactManager does not proceed with the destruction request and notifies the web service that the contact cannot be destroyed.

See also

- *Configuration Guide*

PersonalDataDestructionAPI web service methods

PersonalDataDestructionAPI provides the following methods:

requestContactRemovalWithABUID(addressBookUID : String, requesterID: String) : String

A request to destroy a contact by AddressBookUID. This method is implemented as an asynchronous process that uses work queues.

requestContactRemovalWithPublicID(contactPublicID : String, requesterID: String) : String

A request to destroy a contact by PublicID. This method is implemented as an asynchronous process that uses work queues.

doesABUIDExist(addressBookUID: String): boolean

Uses translateABUIDToPublicIDs as defined in the class that implements the PersonalContactDestroyer interface.

doesContactWithPublicIDExist(publicID: String): boolean

currentDestructionRequestStatusByRequestID(uniqueRequestID : String): DestructionRequestStatus

destroyUser(username : String) : boolean

Given a username, verifies the existence of the credential and the user. This method is a synchronous destruction request that does not involve work queues.

- If the credential exists and the user does not, then the method obfuscates the credential and logs that the user does not exist.
- If both credential and user exist, the method obfuscates the user, which obfuscates both the credential and the UserContact object.
- The method returns a boolean value which is true if the user destruction succeeded, or false otherwise.

See also

- *Configuration Guide*

Lifecycle of a personal data destruction request

The lifecycle of a contact removal request depends on the method that the external system calls to start the request. The lifecycle, also called an *asynchronous* personal data destruction request, is started by a call either to `requestContactRemovalWithABUID` or `requestContactRemovalWithPublicID`. For these two web service method calls, the external system has either the AddressBookUID or the PublicID of the contact whose data to be destroyed. The destroy action performed is defined in the ClaimCenter implementation of the `PersonalDataDestruction` plugin interface.

Note: The `destroyUser` method is synchronous and initiates obfuscation. It does not use work queues, and therefore does not participate in the personal data destruction request lifecycle.

If the web service determines that the request is an existing one, it adds the specified `requesterID` value to the existing destruction request and does not start a new request.

If the web service determines that the request is a new one, the web service:

1. Does the following depending on whether the request is for an AddressBookUID or PublicID:

- If the web service call was to `requestContactRemovalWithABUID`, the web service:
 - Creates a `PersonalDataDestructionRequest` object for the `AddressBookUID`.
 - Adds `PersonalDataContactDestructionRequest` objects for all related `PublicID` values, which are obtained from a call to the `PersonalDataDestroyer` implementation.
 - If the web service call was to `requestContactRemovalWithPublicID`, the web service:
 - Creates a `PersonalDataContactDestructionRequest` object for the `PublicID`.
 - Adds a `PersonalDataDestructionRequest` object if there is a related `AddressBookUID` obtained from a call to the `PersonalDataDestroyer` implementation.
2. Adds a `PersonalDataDestructionRequester` object using `requesterID`.
3. The `DestroyContactForPersonalData` work queue checks for requests in the `ReadyToAttemptDestruction` category, status `New` or `ReRun`, and calls the `Destroyer`.

The class `PersonalDataContactDestructionWorkQueue`, which implements this work queue, calls the following method:

```
PersonalDataDestructionController.destroyContact(contactPurgeRequest)
```

- If the request status is in the `DestructionStatusFinished` category, the queue marks the date of destruction for the contact destruction request.
 - If the request status is `ManualInterventionRequired`, you must implement code that notifies the data protection officer. That user must determine what to do and then set the status to `ReRun` so the `DestroyContactForPersonalData` work queue can run it again.
4. The `NotifyExternalSystemForPersonalData` work queue looks at all `PersonalDataContactDestructionRequest` objects that are associated with a `PersonalDataDestructionRequest`. If they all have a status in the category `DestructionStatusFinished`, the work queue does the notification.
5. The `NotifyExternalSystemForPersonalData` work queue notifies the external system by using `PersonalDataDestructionRequester` objects. As part of this notification, the work queue calls the `PersonalDataDestruction` plugin method `notifyExternalSystemsRequestProcessed`.
6. The `RemoveOldContactDestructionRequest` work queue removes all requests that satisfy both of the following criteria:
- The date obtained by adding the value of the configuration parameter `ContactDestructionRequestAgeForPurgingResults` to the value of `PersonalDataContactDestructionRequest.purgedDate` is less than or equal to today's date.
 - The `PersonalDataContactDestructionRequest` object has a typecode that is in the `DestructionStatusFinished` category.

Personal data destruction request entities

Three kinds of entities are created when a request is made to purge a contact:

PersonalDataDestructionRequest

Holds the `AddressBookUID` and information regarding the parts and result of this destruction request.

PersonalDataContactDestructionRequest

Holds the `PublicID` of the Contact and its current destruction status.

PersonalDataDestructionRequester

Holds the external system `requesterID` that requested the purge and a unique ID associated with the request assigned by ClaimCenter.

Example of a request made with AddressBookUID

If the call is made to the web service method `requestContactRemovalWithABUID`, and two contacts have the same `AddressBookUID`, the destruction request causes the following instances to be created:

- One `PersonalDataDestructionRequest`
- Two `PersonalDataContactDestructionRequest` objects, one for each `PublicID` linked to the `AddressBookUID` request
- One `PersonalDataDestructionRequester`

If the `AddressBookUID` is not found, an exception is thrown.

If an existing destruction request for `AddressBookUID` has `AllRequestFulfilled` equal to `true`, then the external system is notified that destruction has already finished.

Example of a request made with PublicID

If two calls are made to the web service method `requestContactRemovalWithPublicID` for the same `PublicID`, the destruction request would create:

- One `PersonalDataDestructionRequest`
- One `PersonalDataContactDestructionRequest`
- Two `PersonalDataDestructionRequester` objects

If the `PublicID` is not found, an exception is thrown.

If an existing destruction request with `PublicID` has `AllRequestFulfilled` on the `PersonalDataDestructionRequest` equal to `true`, then the external system is notified that destruction has already finished.

TypeLists for status of personal destruction workflow

The personal destruction workflow uses typecodes to indicate various statuses. These typecodes are defined in three typelists, `ContactDestructionStatus`, `DestructionRequestStatus`, and `ContactDestructionStatusCategory`.

ContactDestructionStatus

When a new contact destruction request is started, the initial status of the destruction object is `New`. These status values are defined in the `ContactDestructionStatus` typelist. After a destruction attempt is made on the contact, the destroyer is expected to return a status corresponding to how much it was able to destroy:

New

The initial status of the destruction object when a new contact destruction request is started.

NotDestroyed

Nothing could be destroyed.

Partial

Some data was destroyed.

Not returned in the base configuration of ClaimCenter.

Completed

Everything requested was destroyed.

ManualIntervention

There was an error. This status enables inspection by the Data Protection Officer and must be set before setting `ReRun`.

Not returned in the base configuration of ClaimCenter.

In the base configuration, ClaimCenter provides code that notifies the Data Protection Officer in the plugin class that implements `PersonalDataDestruction`. Additionally, after the Data Protection Officer takes action, the method `PersonalDataDestructionController.requeueContactRemovalRequestWithPublicID` must be called. You must configure a way to make that method call, such as a button in the ClaimCenter user interface.

ReRun

Enables another attempt at destruction.

DestructionRequestStatus

You can retrieve the status of the entire destruction request through the `Status` property on the request itself. These statuses are defined in the `DestructionRequestStatus` typelist. The general status of the entire destruction request can be:

DoesNotExist

The object to be destroyed does not exist.

Unprocessed

Everything is still in the New status.

InProgress

All other combinations of contact destruction statuses.

Finished

Everything is in a final state.

ContactDestructionStatusCategory

Every `ContactDestructionStatus` typecode except `ManualInterventionRequired` has one or more categories.

DestructionStatusNotProcessed

Indicates that the request has not gone through the destruction process.

ReadyToAttemptDestruction

Labels the contact purge request as being ready to be sent to the destroyer.

DestructionStatusFinished

Indicates that the request has finished the destruction process.

ReadyToBeNotified

Labels the request as ready for notification of the external system.

New and ReRun are under the category `ReadyToAttemptDestruction`.

New also is included in the category `DestructionStatusNotProcessed`.

Partial, NotDestroyed, and Completed are under both the category `DestructionStatusFinished` and the category `ReadyToBeNotified`.

Work queues used in personal data destruction

Guidewire provides the following work queues for use in personal data destruction:

- `ArchiveReferenceTrackingSync`
- `DestroyContactForPersonalData`
- `NotifyExternalSystemForPersonalData`
- `RemoveOldContactDestructionRequest`

See the *Administration Guide* for information on these work queues.

PersonalDataDestructionController class

This class handles the complete lifecycle of the asynchronous destruction process after a destruction request has been made through the `PersonalDataDestructionAPI` web service. This class attempts to destroy the contact and notify the requesters that the contact has been destroyed.

The class provides the following methods relating to the lifecycle:

notifyRequesterDestructionRequestHasFinished(destructionRequester: PersonalDataDestructionRequester)

Takes a requester and notifies the external system that the request with related `AddressBookUID` and `PublicID` values was processed and finished.

destroyContact(contactDestructionRequest: PersonalDataContactDestructionRequest)

Called by `PersonalDataContactDestructionWorkQueue`, the class that implements the `DestroyContactForPersonalData` work queue, when attempting destruction. Uses the class that implements the `PersonalDataDestroyer` interface to destroy the contact, and returns a `ContactDestructionStatus`. Verifies status and sets appropriate `PersonalDataContactDestructionRequest` and `PersonalDataDestructionRequest` attributes.

requeueContactRemovalRequestWithPublicID(publicID : String, bundle : Bundle)

Sets purge request status to `ReRun` after manual intervention, enabling the purge request to be reattempted for destruction.

See also

- “Data destruction web service” on page 132
- *Configuration Guide*

The Profiler API web service

The `ProfilerAPI` web service enables or disables the ClaimCenter profiler system for various components, such as batch processes or web services. The `ProfilerAPI` is located in the `gw.wsi.pl` package.

Batch process and work queue profiling

The `setEnableProfilerForBatchProcess` method enables or disables the profiling of a type of batch process.

```
setEnableProfilerForBatchProcess(enable : boolean,
                                 type : String,
                                 hiResTime : boolean,
                                 enableStackTracing : boolean,
                                 enableQueryOptimizerTracing : boolean,
                                 enableExtendedQueryTracing : boolean,
                                 dbmsCounterThresholdMs : int,
                                 diffDbmsCounters : boolean)
```

The method accepts the following arguments.

- `enable` - Boolean value specifying whether to enable (`true`) or disable (`false`) the profiler
- `type` - The type of batch process to profile. The argument value must be a `String` representation of a typecode from the `BatchProcessType` typekey.
- `hiResTime` - Boolean value specifying whether to use the high-resolution clock for the profiler timing. The high-resolution clock is available only on Windows-based systems.
- `enableStackTracing` - Boolean value specifying whether to allow stack tracing. Enabling stack tracing can significantly slow the execution of the batch process. The argument's value is ignored if the `enable` argument is `false`.
- `enableQueryOptimizerTracing` - Boolean value specifying whether to allow query optimizer tracing. Enabling query optimizer tracing can significantly slow the execution of the batch process. The argument's value is ignored if the `enable` argument is `false`.

- `enableExtendedQueryTracing` - Boolean value specifying whether to allow query tracing. Enabling query tracing can significantly slow the execution of the batch process. The argument's value is ignored if the `enable` argument is `false`.
- `dbmsCounterThresholdMs` - The maximum number of milliseconds a database operation can take before generating a report using DBMS counters. To disable reporting, specify a value of zero.
- `diffDbmsCounters` - Boolean value specifying whether to diff DBMS counters. The argument's value is ignored if the `enable` argument is `false` or the `dbmsCounterThresholdMs` argument is zero.

The method enables or disables the profiler for the specified type of batch process. If profiling is enabled, the operation is configured according to the specified argument values.

The method has no return value.

The `setEnableProfilerForWorkQueue` method enables or disables the profiling of work queues associated with a specified type of batch process.

```
setEnableProfilerForWorkQueue(enable : boolean,
                               type : String,
                               hiResTime : boolean,
                               enableStackTracing : boolean,
                               enableQueryOptimizerTracing : boolean,
                               enableExtendedQueryTracing : boolean,
                               dbmsCounterThresholdMs : int,
                               diffDbmsCounters : boolean)
```

The method accepts the same arguments as the `setEnableProfilerForBatchProcess` method.

The method has no return value.

The `setEnableProfilerForBatchProcessAndWorkQueue` method enables or disables the profiling of both a type of batch process and the work queues associated with it.

```
setEnableProfilerForBatchProcessAndWorkQueue(enable : boolean,
                                             type : String,
                                             hiResTime : boolean,
                                             enableStackTracing : boolean,
                                             enableQueryOptimizerTracing : boolean,
                                             enableExtendedQueryTracing : boolean,
                                             dbmsCounterThresholdMs : int,
                                             diffDbmsCounters : boolean)
```

The method accepts the same arguments as the `setEnableProfilerForBatchProcess` method.

The method has no return value.

[Message destination profiling](#)

The `setEnableProfilerForMessageDestination` method enables or disables the profiling of a messaging destination.

```
setEnableProfilerForMessageDestination(enable : boolean,
                                       destinationID : int,
                                       hiResTime : boolean,
                                       enableStackTracing : boolean,
                                       enableQueryOptimizerTracing : boolean,
                                       enableExtendedQueryTracing : boolean,
                                       dbmsCounterThresholdMs : int,
                                       diffDbmsCounters : boolean)
```

The method accepts the same arguments as the `setEnableProfilerForBatchProcess` method, except for the `destinationID` argument, which replaces the `type` argument. The `destinationID` argument specifies the unique numeric ID of the message destination to enable or disable profiling for.

The method has no return value.

Startable plugin profiling

The `setEnableProfilerForStartablePlugin` method enables or disables the profiling of a startable plugin.

```
setEnableProfilerForStartablePlugin(enable : boolean,
                                    pluginName : String,
                                    hiResTime : boolean,
                                    enableStackTracing : boolean,
                                    enableQueryOptimizerTracing : boolean,
                                    enableExtendedQueryTracing : boolean,
                                    dbmsCounterThresholdMs : int,
                                    diffDbmsCounters : boolean)
```

The method accepts the same arguments as the `setEnableProfilerForBatchProcess` method, except for the `pluginName` argument, which replaces the `type` argument. The `pluginName` argument specifies the unique `String` name of the startable plugin to enable or disable profiling for.

The method has no return value.

Web service profiling

The `setEnableProfilerForWebService` method enables or disables the profiling of a web service.

```
setEnableProfilerForWebService(enable : boolean,
                               serviceName : String,
                               operationName : String,
                               hiResTime : boolean,
                               enableStackTracing : boolean,
                               enableQueryOptimizerTracing : boolean,
                               enableExtendedQueryTracing : boolean,
                               dbmsCounterThresholdMs : int,
                               diffDbmsCounters : boolean)
```

The method accepts the same arguments as the `setEnableProfilerForBatchProcess` method, except for the replacement of the `type` argument with the following arguments to specify a web service.

- `serviceName` - Specifies the web service name
- `operationName` - Specifies the web service operation

The method has no return value.

Web session profiling

The `setEnableProfilerForSubsequentWebSessions` method enables or disables the profiling of web sessions.

```
setEnableProfilerForSubsequentWebSessions(name : String,
                                         enable : boolean,
                                         hiResTime : boolean,
                                         enableStackTracing : boolean,
                                         enableQueryOptimizerTracing : boolean,
                                         enableExtendedQueryTracing : boolean,
                                         dbmsCounterThresholdMs : int,
                                         diffDbmsCounters : boolean)
```

The method accepts the same arguments as the `setEnableProfilerForBatchProcess` method, except for the replacement of the `type` argument with the `name` argument. The `name` argument specifies the `String` text to use to identify the profiler. Any text can be used. Unlike the other methods in the `ProfilerAPI` web service, the `name` argument precedes the `enable` argument.

The profiling of web sessions is intended for development environments and is not recommended for production environments.

The method has no return value.

Server state web service

The `ServerStateAPI` web service provides information about the ClaimCenter server. Use this web service to check that the server is at the required run level before running other web services.

This class has a single method, `getServerState` that returns a `ServerStateInfo` object. From Gosu, you can access this object as the read-only `ServerState` property. This object has the following properties:

`RunLevelCode`

An `int` value that is the code for the current run level of the server

`RunLevelName`

A `String` value that is the name of the current run level of the server

`RunLevelOrdinal`

An `int` value that is the ordinal value of the current run level of the server

`ServerId`

A `String` value that is the identifier of the server

`StartupException`

A `String` value that identifies an exception that occurred at server start up

System tools web service

The `SystemToolsAPI` interface provides a set of tools that are always available, even if the server is set to maintenance run level. For servers in clusters, system tools API methods execute only on the server that receives the request. For the complete set of methods in the system tools API, refer to the Gosu implementation class in Guidewire Studio™.

Getting and setting the run level

The following code statement uses the `SystemToolsAPI` interface to retrieve the server run level.

```
runlevelString = systemTools.getRunLevel().getValue();
```

The following code statement sets the server run level.

```
systemTools.setRunLevel(SystemRunlevel.GW_MAINTENANCE);
```

Sometimes you want a more lightweight way than SOAP APIs to determine the run level of the server. During development, you can check the run level by using your web browser.

To check the run level, call the ping URL on a ClaimCenter server using the following syntax.

```
http://server:port/cc/ping
```

An example command line is shown below.

```
http://ClaimCenter.mycompany.com:8080/cc/ping
```

If the server is running, the browser returns a short HTML result with a single ASCII character to indicate the run level. If you are checking whether the server is running, any return value from the ping URL proves the server is running. If the browser returns an HTTP or browser error, the server is not running.

To determine the current run level, examine the contents of the HTTP result for an ASCII character.

ASCII character in ping URL result	Run level
No response to the HTTP request	Server not running
ASCII character 30, the Record Separator character	DB_MAINTENANCE
ASCII character 40, the character "("	MAINTENANCE
ASCII character 50, the character "2"	MULTIUSER

ASCII character in ping URL result	Run level
ASCII character 0. A null character result might not be returnable for some combinations of HTTP servers and <code>GW_STARTING</code> clients.	

Getting server and schema versions

You can use the `SystemToolsAPI` web service method `getVersion` to get the current server and schema versions.

The method returns a structure that contains the following properties.

```
AppVersion  
    Application version as a String  
PlatformVersion  
    Platform version as a String  
SchemaVersion  
    Schema version as a String  
CustomerVersion  
    Customer version as a String
```

Clustering tools

There are several related `SystemToolsAPI` web service methods for managing clusters of servers. Call these methods from an external system to get information about the cluster and manage failure conditions.

Get cluster state

To get a list of all nodes in the cluster, their roles, and what distributed components they run, call the `SystemToolsAPI` web service method `getClusterState`.

It takes no arguments and returns an object of type `ClusterState`, which encapsulates the following properties.

- `Members` – Cluster members, as a list of `ClusterMemberState` objects. Each `ClusterMemberState` has the following properties.
 - `ServerId` – Server unique ID
 - `InCluster` – Boolean value that indicates if the server is successfully in the cluster
 - `RunLevel` – Run level
 - `Roles` – List of roles. Each role is a `String` value
 - `ServerStarted` – Boolean value that indicates if the server is started
 - `LastUpdatedTime` – Last time when the server updated its status, as a `Date` object
 - `PlannedShutdownInitiated` – Date start of an initiated shutdown, or `null` if none
 - `PlannedShutdownTime` – Date of a planned shutdown, or `null` if none
 - `BgTasksStopped` – Date value that indicates when background tasks stopped, or `null` if they have not stopped
 - `UserSessions` – user sessions
- `Components` – Resources used by this server member. This is a list of `ComponentInfo` objects, each one of which represents a lease on a resource, such as a messaging destination. The `ComponentInfo.Type` property indicates the type of reserved resource, as a value from the `ComponentType` enumeration: `WORK_QUEUE`, `SYSTEM`, `BATCH_PROCESS`, `STARTABLE_PLUGIN`, or `MESSAGE_DESTINATION`. For other fields on `ComponentInfo`, view the class in Studio. The `ComponentInfo.UniqueId` property contains the unique ID for this component.
- `UnassignedComponents` – Unassigned resources as a list of `ComponentInfo` objects.

Clean and release resources after node failure

To clean and release resources, such as batch processes, plugins, message destinations, that are reserved by a specified node, call the `SystemToolsAPI` web service method `nodeFailed`. Be aware, however, that if the node is still running, this method can have a dangerous impact to data integrity and proper application behavior.

The method takes the following arguments.

- A `String` server ID
- A `boolean` argument called `evenIfInCluster`, which specifies whether to consider the node as failed if it is still in the cluster.

You must ensure that the server referenced by the server ID is actually stopped if you set the `evenIfInCluster` option to true. ClaimCenter does not prevent you from using this option if the server is still running. However, the option overrides an important safety check on the server. It can produce unexpected and possibly negative results if the server is running. Use the `evenIfInCluster` option only if both of the following are true: (1) The server in question is no longer running and (2) the standard operation of the `nodeFailed` method failed due to the server retaining its cluster membership.

The method has no return value.

Complete component failure

To complete a failed failover for a reservable resource, such as a messaging destination, call the `SystemToolsAPI` web service method `completeFailedFailover`.

```
completeFailedFailover(componentType : ComponentType, componentId : String)
```

The `componentType` argument references a `ComponentType` enumeration value. Supported values are `WORK_QUEUE`, `SYSTEM`, `BATCH_PROCESS`, `STARTABLE_PLUGIN`, `MESSAGE_DESTINATION`, and `MESSAGE_PREPROCESSOR`.

The `componentId` argument specifies the component's unique `String` ID.

The method has no return value.

Table import web service

The `TableImportAPI` web service provides tools for importing non-administrative data from staging tables into operational tables in the ClaimCenter database. The `table_import` command-line tool wraps this API to provide a straightforward way to use the web service. To import data from external sources, you first write a conversion tool to populate the staging tables. Next, you use methods from this web service to check the integrity of those tables. You cannot load data until these methods report no errors. Then, use one of the load methods to load data from the staging tables into operational tables. ClaimCenter uses bulk data import commands to perform the load.

Most of the methods for this web service require the server to be at the `MAINTENANCE` run level. Before you use those methods, start the server at the `MAINTENANCE` run level. Alternatively, use the `SystemToolsAPI` web service `setRunLevel` method to set the server run level to `MAINTENANCE`. After the table import command completes, you can set the server run level to `MULTIUSER`.

Some tasks are available as batch operations. Methods for batch operations return a process ID that you can check for completion of the task. Use these methods to avoid holding a web transaction open for an indefinite period of time. To check for completion of the batch process, you use the `MaintenanceToolsAPI` web service `batchProcessStatusByID` method. The method returns a `ProcessStatus` object. To check the progress of the batch process, use the following `boolean` properties.

- `Starting` – The batch process is initializing.
- `Executing` – The batch process is running.
- `Complete` – The batch process has finished.

- Success – The batch process completed successfully.

The `ProcessStatus` object also provides more detailed information about the batch process. For example, the following code checks the status of an integrity check batch process.

```
var tiApi = new TableImportAPI()
var mtApi = new MaintenanceToolsAPI()
// Authenticate the APIs here
...
var tiBatchProcess = tiApi.integrityCheckStagingTableContentsAsBatchProcess(false, false, false, 1)
...
try {
    // Check the status
    var status = mtApi.batchProcessStatusByID(new () { :Pid = tiBatchProcess.Pid })
    if (status.StartingOrExecuting) {
        ...
    } else {
        print("Started at ${status.StartDate}, completed at ${status.DateCompleted}")
        if (status.Success) {
            print("Ran to completion, completing ${status.OpsCompleted} operation(s) with " +
                "${status.FailedOps} failure(s);")
        } else {
            print("Failed to run to completion: ${status.FailureReason}. " +
                "Completed ${status.OpsCompleted} operation(s) " +
                "with ${status.FailedOps} failure(s) before terminating")
        }
    }
} catch (e) {
    print("Failed to find process status: ${e.getClass()} " + e.Message)
}
```

For staging table actions that you can request from remote systems, refer to the following table.

Action	TableImportAPI method and Description
Clear data from staging tables in the database	<code>clearStagingTables</code> Before using a conversion tool to load new data from an external source to staging tables in the database, you typically clear the existing data from the staging tables. This synchronous method returns nothing.
Clear data from the error table in the database	<code>clearErrorTable</code> Before running integrity checks on data in the staging tables, you typically clear existing load errors from the database. This synchronous method returns nothing.
Clear data from the exclusion table in the database	<code>clearExclusionTable</code> Before running integrity checks on data in the staging tables, you typically clear existing records from the exclusion table. This synchronous method returns nothing.
Perform checks on the staging tables to ensure that the data will load into operational tables	<code>integrityCheckStagingTableContentsAsBatchProcess</code> Before you load staging table data into operational tables, your staging table data must pass the integrity checks that this method performs for ClaimCenter. This method creates entries in the error table for errors that it detects. This method returns a process ID that you can check for completion of the action.
Prepare to remove staging tables rows that cause integrity checks to fail	<code>populateExclusionTableAsBatchProcess</code> You can choose to remove the rows from staging tables that cause integrity checks to fail if you do not want to correct the errors. Populating the exclusion table marks these rows for removal. This method returns a process ID that you can check for completion of the action.
Delete data specified by the exclusion table from the staging tables	<code>deleteExcludedRowsFromStagingTablesAsBatchProcess</code> Before rerunning integrity checks on data in the staging tables, you can remove records that contain errors from the staging tables. This method returns a process ID that you can check for completion of the action.
Encrypt data in the staging tables before loading the data into operational tables	<code>encryptDataOnStagingTablesAsBatchProcess</code> If you have database columns that are encrypted, you must encrypt the data values in the staging tables before you load the data into operational tables. This method returns a process ID that you can check for completion of the action.

Action	TableImportAPI method and Description
Update database statistics on staging tables	<code>updateStatisticsOnStagingTablesAsBatchProcess</code> To ensure the database performs the load operations effectively, update the statistics on the staging tables before loading the data into operational tables. For an Oracle database, this method also updates the indexes on the staging tables. This method returns a process ID that you can check for completion of the action.
Load data from staging tables into operational tables after performing checks on the staging tables	<code>integrityCheckStagingTableContentsAndLoadSourceTablesAsBatchProcess</code> Prior to loading the data from the staging tables to operational tables, this method reruns integrity checks. If the checks pass, this method loads all staging table data into operational tables. This process performs any required data conversions. For example, placeholder values for foreign keys are converted to actual key values. If this process encounters an unrecoverable error, it reverts the entire data load. If the load process succeeds, it clears all data from the staging tables. This method returns a process ID that you can check for completion of the action.
Load zone data from staging tables into operational tables after performing checks on the staging tables	<code>integrityCheckZoneStagingTableContentsAndLoadZoneSourceTables</code> This method performs the same action as <code>integrityCheckStagingTableContentsAndLoadSourceTables</code> , but loads only zone data. This synchronous method returns the result of the load operation.
Get information about completed staging table operations	<code>getLoadHistoryReportsInfo</code> This synchronous method returns a String array containing summary information about the most recent calls to the TableImportAPI web service. The array members are in reverse chronological order. The number of operations is specified by the argument to this method. An argument of 0 returns all available history.

Template web service

The TemplateToolsAPI web service enables an external system to list and validate document, email, and note templates. The service also enables fields in a template descriptor file to be imported from CSV (comma-separated value) text files.

List templates

The following web service methods retrieve a list of templates on the server. The list can be used to sanity-check the arguments to the web service's validation methods.

```
listDocumentTemplates() : String
listEmailTemplates() : String
listNoteTemplates() : String
```

Each method returns a human-readable string that describes the templates available on the server.

Validate templates

The following methods validate either a single or all templates. Each method processes a specific type of template.

```
validateDocumentTemplate(templateID : String) : String
validateAllDocumentTemplates(): String

validateEmailTemplate(templateFileName : String, beanNamesAndTypes : NameTypePair[]) : String
validateAllEmailTemplates(beanNamesAndTypes : NameTypePair[]) : String

validateNoteTemplate(templateFileName : String, beanNamesAndTypes : NameTypePair[]) : String
validateAllNoteTemplates(beanNamesAndTypes : NameTypePair[]) : String
```

Each method accepts a String file name that specifies the template to validate, such as `ReservationRights.doc`. The email and note methods also accept an array of `NameTypePair` objects. Each `NameTypePair` object contains a parameter name and its type. Each property is specified as a String. The two properties are used to validate the template placeholder.

Each method returns a human-readable string that describes the operations performed, plus any errors, if they occurred.

The following validation tests are performed.

- Checks all Gosu `ContextObject` and `FormField` expressions in the descriptor. `ContextObject` expressions must be defined in terms of the available objects. `FormField` expressions must be defined in terms of either available objects or `ContextObjects`.
- Checks that the `permissionRequired` attribute is valid, if specified.
- Checks that the `default-security-type` attribute is valid, if specified.
- Checks that the `type` attribute is a valid type code, if specified.
- Checks that the `section` attribute is a valid section type code, if specified.

Each validation method has a counterpart that accepts a locale.

```
validateDocumentTemplateInLocale(templateID : String, locale : String) : String  
validateAllDocumentTemplatesInLocale(locale : String) : String  
  
validateEmailTemplateInLocale(templateFileName : String, beanNamesAndTypes : NameTypePair[], locale : String) : String  
validateAllEmailTemplatesInLocale(beanNamesAndTypes : NameTypePair[], locale : String) : String  
  
validateNoteTemplateInLocale(templateFileName : String, beanNamesAndTypes : NameTypePair[], locale : String) : String  
validateAllNoteTemplatesInLocale(beanNamesAndTypes : NameTypePair[], locale : String) : String
```

Import form fields

The `importFormFields` method imports context objects, field groups, and fields from a CSV (comma-separated values) text file into a specified template descriptor file.

```
importFormFields(contextObjectFileContents : String,  
                 fieldGroupFileContents: String,  
                 fieldFileContents : String,  
                 descriptorFileContents : String) : TemplateImportResults
```

The method accepts the following arguments.

- `contextObjectFileContents` - The contents of a CSV file that specifies the context objects to import
- `fieldGroupFileContents` - The contents of a CSV file that specifies the field groups to import
- `fieldFileContents` - The contents of a CSV file that specifies the fields to import
- `descriptorFileContents` - The contents of the descriptor file

The method returns a `TemplateImportResults` object with fields for the newly-imported contents of the descriptor file. The object also contains a human-readable string that describes the operations performed, plus any errors, if they occurred.

Workflow web service

The `WorkflowAPI` web service enables ClaimCenter workflows to be controlled remotely. The built-in `workflow_tools` command-line tool also uses the web service to provide local control of workflows.

Workflow basics

You can invoke a workflow trigger remotely from an external system using the `invokeTrigger` method.

Any time the application detects a workflow error, the workflow sets itself to the state `TC_ERROR`. When this occurs, you can remotely resume the workflow using these APIs.

Refer to the following table for workflow actions that you can request from remote systems.

Action	WorkflowAPI method and Description
Invoke a workflow trigger	<code>invokeTrigger</code>

Action	WorkflowAPI method and Description
	Invokes a trigger key on the current step of the specified workflow, causing the workflow to advance to the next step. This method takes a workflow public ID and a String value that represents the workflow trigger key from the WorkflowTriggerKey typelist. To check whether you can call this workflow trigger, use the isTriggerAvailable method in this interface (see later in this table). This method returns nothing.
Check whether a trigger is available	<code>isTriggerAvailable</code> Check if a trigger is available in the workflow. If a trigger is available, it means that it is acceptable to pass the trigger ID to the invokeTrigger method in this web services interface. This method takes a workflow public ID and a String value that represents the workflow trigger key from the WorkflowTriggerKey typelist. It returns true or false.
Resume a single workflow	<code>resumeWorkflow</code> Restarts one workflow specified by its public ID. This method sets the state of the workflow to TC_ACTIVE. This method returns nothing.
Resume all workflows	<code>resumeAllWorkflows</code> Restarts all workflows that are in the error state. It is important to understand that this only affects workflows currently in the error state TC_ERROR or TC_SUSPENDED. The workflow engine subsequently attempts to advance these workflows to their next steps and set their state to TC_ACTIVE. For each one, if an error occurs again, the application logs the error sets the workflow state back to TC_ERROR. This method takes no arguments and returns nothing.
Suspend a workflow	<code>suspend</code> Sets the state of the workflow to TC_SUSPENDED. If you must restart this workflow later, use the resumeWorkflow method or the resumeAllWorkflows method.
Complete a workflow	<code>complete</code> Sets the state of a workflow specified by its public ID to TC_COMPLETED. This method returns nothing.

Zone data import web service

The ZoneImportAPI web service provides tools for importing zone data from comma-separated values (CSV) files into database staging tables in the ClaimCenter database. The `zone_import` command-line tool wraps the API to provide a straightforward way to use the web service. Guidewire recommends that you use the `zone_import` tool.

After importing the data into staging tables, you use the table import (TableImportAPI) web service to load the staging table data into the operational tables in the database. ClaimCenter uses bulk data import commands to perform the data load.

For zone data actions that you can request from remote systems, refer to the following table.

Action	ZoneImportAPI method and Description
Clear zone data from operational tables in the database	<code>clearProductionTables</code> Before loading new zone data from staging tables to the operational tables in the database, you clear the existing zone data from the operational tables. The method takes an optional two-letter country code. Provide this value if you plan to load zone data for a subset of zones. The method returns nothing.
Clear zone data from staging tables in the database	<code>clearStagingTables</code> Before loading new zone data from comma-separated values (CSV) files to staging tables in the database, you clear the existing zone data from the staging tables. The method takes an optional two-letter country code. Provide this value if you plan to load zone data for a subset of zones. The method returns nothing.

Action	ZoneImportAPI method and Description
Import zone data from CSV files to staging tables in the database	<code>importToStaging</code> Use this method to load new zone data from a CSV file to staging tables in the database. The method takes two <code>String</code> parameters: a two-letter country code and a path to the CSV file. A boolean flag parameter specifies whether to clear data for this country from the staging tables. The method returns the number of rows that the method loaded into staging tables.

Claim-related web services

See also

- “Web services” on page 67

Claim web service APIs and data transfer objects

To manipulate claims and exposures from external systems in general ways, use the `ClaimAPI` WS-I web services. If you need to manipulate claim financials, instead use the separate `ClaimFinancialsAPI` web service.

The ClaimCenter WS-I web services do not support entity data directly as method arguments or return values. Therefore, any web service APIs use a separate data transfer object (DTO) to encapsulate entity data. For example, instead of passing an `Address` entity directly as a method argument, the API might pass a Gosu class called `AddressDTO`, which acts as the DTO. The DTO class contains only the properties in an `Address` entity instance that are necessary for the web service.

If you extend the entity data model with properties that must exist in the web service for sending or receiving data, you must also extend the data transfer object. Add to the set of properties in the corresponding Gosu class with the `DTO` suffix. For example, if you add an important property to the `Address` entity, also add the property to the Gosu class `AddressDTO`.

These methods are for active claims only. If an external system posts any new updates on an archived claim, `ClaimAPI` methods throw the exception `EntityStateException`.

Your web service definition in the WSDL defines a strict programmatic interface to external systems that use your service. The WSDL encodes the structure of all parameters and return values. After moving code into production, be careful not to change the WSDL. For example, do not modify DTOs after going into production or widely distributing the WSDL in a user acceptance testing UAT environment.

See also

- “Claim financials web service (`ClaimFinancialsAPI`)” on page 421

Date- and time-related DTOs

Several Gosu types related to date and time are based on XSD schemas. Examples include `XSDDate` and `XSDDateTime`. These XSD-based types can be used as web service DTOs. To represent a valid date or time, you must specify the relevant time zone in the web service call. Failure to specify a time zone results in an `IllegalStateException`.

The Gosu XSD-based types that require a time zone when used as a DTO are listed below.

- XSDDate
- XSDDateTime
- XSDGDay
- XSDGMonth
- XSDGMonthDay
- XSDGYear
- XSDGYearMonth
- XSDDTime

The Gosu XSD-based types can be constructed from a `java.lang.String` object that conforms to the XML Schema date/time formats specified in the W3C XML Schema specification.

```
construct(s : String)
```

To convert an XSD-based type to a `String`, use the object's `toString` method.

An XSD-based date/time object can also be constructed from a `java.util.Calendar` object.

```
construct(cal : Calendar, useTimeZone : boolean)
```

If the constructor's `useTimeZone` argument is `true`, the constructed object inherits the time zone specified in the `Calendar` object.

Conversely, a Gosu XSD-based date/time object can be converted to a `java.util.Calendar` object by calling the object's `toCalendar` method. `Calendar` fields that are not included in the XSD-based object are set to default values. Two method signatures are supported.

```
toCalendar() : Calendar  
toCalendar(tzDefault : TimeZone) : Calendar
```

If the Gosu object does not specify a time zone value, calling the `toCalendar` method signature with no arguments results in an `IllegalStateException`.

Adding first notice of loss from external systems

You can add a first notice of loss (FNOL) from an external system using the `ClaimAPI` method `addFNOL`.

Never add new claims or exposures to ClaimCenter with the web service APIs while the Financials Calculations batch process is running. This applies to `ClaimAPI` web service methods `addFNOL`, `migrateClaim`, `addExposure`, and `addExposures`. The batch process runs only while the server is in maintenance mode.

If you add an FNOL with `addFNOL`, ClaimCenter performs the following operations.

1. Runs the Loaded rule set.
2. Runs all the normal rule sets, initial reserves, access control lists as for new claims in the New Claim Wizard.

The `addFNOL` method takes exactly two arguments.

- A claim `claimDTO` DTO object, which includes exposures
- A policy `policyDTO` DTO object

The method returns the `String` public ID of the newly-created claim.

Getting a claim from external systems

An external system can get the populated data transfer object (DTO) for a claim if the external system knows the public ID for the claim. Call the `ClaimAPI` web service method `getDtoForClaim` to get the populated DTO for that claim. For example, the following Java code gets the loss cause for a claim.

```
ClaimDTO c = claimAPI.getDtoForClaim("cc:1234");
LossCauseDTO lossCause = c.getLossCause();
```

Importing a claim from XML from external systems

There are two `ClaimAPI` methods to import a claim from XML data from an external system.

- If the format is the ACORD format, use the `importAcordClaimFromXML` method.
- If the format is not the ACORD format, use the `importClaimFromXML` method.

Migrating a claim from external systems

You can migrate an existing claim from another system with the `ClaimAPI` method `migrateClaim`. If you migrate a claim, some normal processing steps for an entirely new claim are unnecessary.

Never add new claims or exposures to ClaimCenter with the web service APIs while the Financials Calculations batch process is running. This applies to `ClaimAPI` web service methods `addFNOL`, `migrateClaim`, `addExposure`, and `addExposures`. The batch process runs only while the server is in maintenance mode.

For example, it is unnecessary to pick a claim number and assign to an adjuster to migrate a claim. ClaimCenter assumes that the regular load rules or setup that ClaimCenter runs for a first notice of loss (FNOL) are unnecessary. Contrast this with the `addFNOL` method.

To migrate a claim, ClaimCenter performs the following operations.

1. Set the claim state to open.
2. Assign the claim to the appropriate user and group.
3. Validate the claim at the `loadsave` level.
4. Commit the claim's data to the database, including all exposures, activities, and other objects creating during the request.
5. Commit the data to the database triggers pre-update rules, double checks validation, and run any appropriate Event Fired rule sets for new entities.

The method takes exactly one argument which is a `ClaimDTO` object. The method returns the `String` Public ID of the newly-created claim.

Getting information from claims/exposures from external systems

The `ClaimAPI` web service provides various straightforward methods to get information from a claims, for example determining if the claim exists, is valid, is flagged, or is open.

The following table summarizes some various simple methods that you can use from external systems.

To do this task	Use this <code>ClaimAPI</code> method	Description
Check if a claim exists	<code>doesExist</code>	Takes a public ID for a claim and determines if the claim exists.

To do this task	Use this ClaimAPI method	Description
	<code>claimsExist</code>	Similar to <code>doesExist</code> , but takes a list of claim public ID String values, not a single public ID.
Check if a claim reached a specified validation level	<code>checkValid</code>	Takes a public ID for a claim and a minimum validation level. Returns <code>true</code> if the claim reached that validation level. Otherwise, returns <code>false</code> .
Check if a claim is flagged	<code>isFlagged</code>	Takes a public ID for a claim, and returns <code>true</code> if the <code>Claim</code> .Flagged value has the value <code>TC_ISFLAGGED</code> .
Get claim state	<code>getClaimState</code>	Returns the code of the claim state (a <code>ClaimState</code> typecode) converted to a string. If the claim state for that claim is undefined, returns <code>null</code> .
Find claim public ID from a claim number	<code>findPublicIDByClaimNumber</code>	Takes a claim number and returns the public ID of the corresponding claim.
Get the claim info for a claim, which corresponds to the <code>ClaimInfo</code> entity.	<code>getClaimInfo</code>	Takes a public ID for a claim, and returns a data transfer object for the <code>ClaimInfo</code> object (<code>ClaimInfoDTO</code>).
Get exposure state	<code>getExposureState</code>	Takes an exposure public ID and returns the exposure state as an <code>ExposureState</code> enumeration, or <code>null</code> if undefined.

Claim bulk validate from external systems

After importing many claims, you can schedule the newly-imported claims for validation, including associated policies and exposures. To use this API, you must know the load command ID from the import request, which is sometimes referred to as a `loadCommandID`.

Bulk validation uses a work queue to spread the work of validating large numbers of claims across multiple threads/nodes. This method does not perform the validation synchronously. This API creates a distributed work item to validate each claim, and starts a batch process. The batch process runs as an independent process, so do not expect the work items to complete before the method returns.

The method returns a process ID value of type `long`. Use the process ID value to query or manipulate the status of the batch process using the methods on `MaintenanceToolsAPI`. For example, to query the batch process, call the `MaintenanceToolsAPI` method `batchProcessStatusByID`. To cancel the batch process, call the `MaintenanceToolsAPI` method `terminateBatchProcessByID`.

By default, no worker instances are configured to run for this process. To perform the actual validation, use the `MaintenanceToolsAPI` web service method `setWorkQueueConfig` to dynamically allocate worker instances.

Bulk load and validate claims

Procedure

1. Load claims with `TableImportAPI.integrityCheckStagingTableContentsAndLoadSourceTables`.
2. From the returned `TableImportResult` object, extract the `loadCommandID`.
3. Call `ClaimAPI.bulkValidate(loadCommandID)` and get the return value process ID value.
4. If you have not yet configured workers, as is the case in the base configuration, configure workers using `MaintenanceToolsAPI` web service. Call the method `setWorkQueueConfig` with work queue name `ClaimValidation`.
5. Notify workers using `MaintenanceToolsAPI` web service method `notifyQueueWorkers` with work queue name `ClaimValidation`.
6. Regularly call the `MaintenanceToolsAPI` method `batchProcessStatusByID` using the returned process ID until the batch process completes.

Previewing assignments from external systems

From an external system, you can preview the assignment choices that ClaimCenter would make for a claim. Call the `ClaimAPI` web service method `previewAssignment`. The method takes a claim public ID and returns an `AssignmentResponse` object that encapsulates the results. The `AssignmentResponse` object includes properties such as `ReviewAssignment`, `UserID`, `GroupID`, `GroupType`, `QueueID`, and `ReviewerID`.

The `previewAssignment` method does not commit to the database any changes made to the claim.

Closing and reopening a claim from external systems

From an external system, you can close or reopen a claim using methods on the `ClaimAPI` web service.

To close a claim, call the `closeClaim` method, and pass it the following arguments.

- Public ID for a claim
- An outcome type, in an `ClaimClosedOutcomeType` enumeration
- A reason, which is an optional `String` value that describes the reason for closing the claim

The method has no return value.

ClaimCenter uses the same logic that governs the Close Claim screen user interface.

To reopen a claim, call the `reopenClaim` method.

- Public ID for a claim
- The reason type for re-opening the claim. The argument is an enumeration of type `ClaimReopenedReason`.
- A reason, which is an optional `String` value that describes the reason for re-opening the claim

Policy refresh from external systems

There are several methods on the `ClaimAPI` web service that manipulate policy data on a claim. However, changing the policy can have many side effects, especially if the old and new policies do not contain the same set of subobjects, such as vehicles. Use these methods with caution.

To refresh the policy on the claim with the latest information from the external policy administration system, call the `refreshPolicy` method. It takes a public ID for a claim and returns no result. The `claim.LossDate` field must be non-null for that claim, or this method throws an `RequiredFieldException` exception.

In some cases, you may want to refresh the policy based on the subset of information in a policy summary object, which corresponds to the `PolicySummary` entity type. If this is the case, instead use the `ClaimAPI` web service method `setPolicy`. It takes a data transfer object for a policy summary encapsulated in a `PolicySummaryDTO` object. Add any necessary properties to the `PolicySummaryDTO` parameter, which ClaimCenter uses to populate a request to your `IPolicySearchAdapter` plugin implementation. The set of required fields is only the set of fields that `IPolicySearchAdapter` needs. This set of fields may be fewer than the set of fields necessary to display the summary in the user interface.

The method has no return value.

See also

- “Policy search plugin” on page 472

Add user permissions on a claim from external systems

To grant a user permissions on a claim from an external system, call the `ClaimAPI` method `giveUserPermissionsOnClaim`. It takes the following arguments.

- A claim public ID
- A user public ID
- An array of access permission types to set, as the type `ClaimAccessType`

The method has no return value.

The new permissions are additive, any existing permissions remain if already existing. The method never removes permissions. If you pass an access permission type that the user already has, there is no effect for that permission and the method does not throw an exception.

Archiving claims from external systems

To schedule a claim for archiving, there are two methods in the `ClaimAPI` web service.

- To schedule by claim number, call `scheduleForArchive`. The method takes an array of claim number `String` values.
- To schedule by public ID, call `scheduleForArchiveByPublicId`. The method takes an array of public IDs for claims.

For each claim, ClaimCenter confirms that the claim is closed, and then schedules the claim for archiving by creating a high priority work item that the archiving work queue processes.

In both cases, the claim does not archive immediately. An asynchronous batch process archives the claim the next time that the process is executed. Therefore, do not assume that claims are archived after these method calls return.

There is a race condition that can affect these calls. If a claim to be archived references a newly created administrative object, such as a new user, there is a chance the archiving of the claim fails. This failure can happen if the new administrative object has not yet been copied to the archiving database. This failure is a rare edge case because most claims to be archived are old, closed claims that have been unaltered for a long time. The chances of hitting this race condition can be minimized by explicitly running the archive batch process before calling this method. However, this workaround is resource-intensive and is not recommended as a general practice.

The methods throw the `SOAPException` exception if claims cannot be scheduled for archive because they cannot be found, are closed, or because an archive work item could not be created. If any of the claims is not found or not closed, then the call fails before attempting to archive any other claims. However, if all claims are present and closed, it is possible, though very unlikely, for ClaimCenter to create some work items successfully and for other work items to fail.

The following Gosu web service client example shows the variant that uses public IDs.

```
claimPublicIDs = { "abc:1234","abc:5678" }
claimAPI.scheduleForArchiveByPublicId(claimPublicIDs)
```

These methods have no return value.

Restoring archived claims from external systems

To restore archived claims, call the `ClaimAPI` web service method `restoreClaims`. The method takes the following arguments:

- An array of `String` holding public ID values of the claims to restore
- A `String` comment typically indicating the reason for restoring the claims

The method returns a `String` indicating the public IDs of the claims that were processed and skipped.

The `restoreClaims` method is unusual in the `ClaimAPI` web service because it does not contain the `@WsiCheckDuplicateExternalTransaction` annotation. Not using this annotation allows the method to avoid using or enforcing transaction IDs and to restore each claim in a separate bundle commit. Multiple bundle commits with the same transaction ID are incompatible with the `@WsiCheckDuplicateExternalTransaction` annotation.

The following example restores two archived claims.

```
String[] restored;
String[] claimPublicIDs = { "abc:1234", "abc:5678" };
// restore the claims:
restored = claimAPI.restoreClaim(claimPublicIDs, "Policy system needs to restore claim to view it");
```

See also

- “Checking for duplicate external transaction IDs” on page 104

Managing activities from external systems

The ClaimAPI web service provides several activity-related methods.

Completing or skipping activities

To complete an activity, call the ClaimAPI web service method `completeActivity`.

To skip an activity, call the `skipActivity` method.

Both methods accept an argument of activity’s String public ID and have no return value. Both methods also require the claim to have the permissions `VIEW_CLAIM` and `CREATE_ACTIVITY`. Finally, both methods run the standard rule sets for skipped or completed activities, and set the activity properties `CloseDate` and `CloseUser`.

Adding activities from DTO

From an external system, you can create a new activity and enter all properties in the data transfer object (DTO). Set the properties on the `ActivityDTO` object and pass it as an argument to the `createActivity` method.

```
var claimPublicID = claimAPI.createActivity(activityDTO)
```

The following properties on the `ActivityDTO` object must be non-null: `ClaimID`, `ActivityPatternID`.

The method returns the String public ID of the newly-created activity.

Adding activities from an activity pattern

To create an activity from an activity pattern, call the `addActivityFromPattern` method. The method takes the following two parameters.

- The claim public ID, which cannot be null
- The activity pattern public ID, which cannot be null

The activity pattern must be from the list of activity patterns for the given claim that meet the following criteria. Otherwise, the method throws an `EntityStateException`.

- If the claim is closed, then the activity pattern must be available to closed claims.
- The loss type on the activity pattern must be null or must match the loss type on the claim.

The new activity is initialized with the following fields from the activity pattern.

- `Pattern`, `Type`, `Subject`, `Description`, `Mandatory`, `Priority`, `Recurring`, and `Command`

ClaimCenter calculates the activity target using the following fields in the pattern.

- `targetStartPoint`, `TargetDays`, `TargetHours`, and `TargetIncludeDays`

The escalation date of the activity is calculated using the following fields in the pattern.

- `escalationStartPoint`, `EscalationDays`, `EscalationHours`, and `EscalationIncludeDays`

If those fields are not included in the activity pattern, then ClaimCenter does not set the target or escalation date. If the target date is calculated to be after the escalation date, then the target date is set to be the same as the escalation date.

The claim public ID of the activity is set to the given claim public ID, and the exposure public ID is set to `null`. The previous user public ID property is updated with the current web service authenticated user.

ClaimCenter assigns the newly-created activity to a group and/or user using the assignment engine. Finally, ClaimCenter persists the activity in the database. The method returns public ID of the newly-created activity.

The following Java statement calls the `addActivityFromPattern` method.

```
claimAPI.addActivityFromPattern("abc:123", "abc:emailnote1");
```

Get activity patterns

To get the data transfer object (DTO) for an activity pattern object from its activity pattern code, call the `ClaimAPI` method `getActivityPatternDataForCode`. It takes the code as a `String` value and returns the `ActivityPatternDTO` instance.

To get the DTO for an activity pattern object from its public ID, call the `ClaimAPI` method `getActivityPatternData`. It accepts the `String` public ID and returns the `ActivityPatternDTO` instance.

Adding a contact from external systems

To add a contact to the ClaimCenter database from an external system, call the `createContact` method. The method accepts a `ContactDTO` object and returns the public ID of the newly-created `Contact` entity instance.

Adding a document from external systems

To add a document to the ClaimCenter database from an external system, call the `ClaimAPI` web service method `createDocument`. The method takes a `DocumentDTO` and returns the public ID of the newly-created `Document` entity instance.

The `createDocument` method requires that the `DocumentDTO.ClaimPublicID` property refers to a valid claim. A document is always related to a `Claim` object. A document can also be optionally related to at most one subobject on that same claim, such as a `ClaimContact`, `Exposure`, or `Matter` object. The properties on `DocumentDTO` reference these objects as `ClaimContactID`, `ExposureID`, and `MatterID`.

Adding an exposure from external systems

The `ClaimAPI` web service contains several methods to create exposures on a claim.

The `ClaimAPI` method `addExposure` creates one new exposure and associates it with a claim. If the state of the claim is `open`, then the method runs the standard save and setup processes on the new exposure. If the state of the claim is `draft`, this method merely sets the exposure order on the claim.

The `addExposure` method accepts a single argument referencing the `ExposureDTO` object to add. The `ExposureDTO.ClaimID` property must contain the public ID of the associated claim.

The method returns a public ID `String` that contains the identifier of the newly created exposure.

To add multiple exposures to the same claim in one method call, use the `addExposures` method instead of `addExposure`. The method accepts the following arguments.

- The claim public ID
- An array of `ExposureDTO` objects. The `ExposureDTO.ClaimID` property must either be `null` or match the claim public ID passed into the method.

It is unsupported to add exposures to multiple claims in the same call to `addExposures`.

The `addExposures` method returns an array of `String` public IDs for the newly created `Exposure` entity instances.

Never add new claims or exposures to ClaimCenter with the web service APIs while the Financials Calculations batch process is running. This applies to `ClaimAPI` web service methods `addFNOL`, `migrateClaim`, `addExposure`, and `addExposures`. The batch process runs only while the server is in maintenance mode.

Getting an exposure from external systems

To get an exposure from an external system, call the `ClaimAPI` web service method `getDtoForExposure`. The method accepts a public ID argument that references the exposure and returns the retrieved `ExposureDTO` object.

Closing and reopening an exposure from external systems

The `ClaimAPI` web service contains methods to close and reopen an exposure.

To close an exposure, call the `closeExposure` method. The method accepts three arguments.

- An exposure public ID
- An outcome enumeration from the `ExposureClosedOutcomeType` typelist
- A `String` that describes why you are closing the exposure

The method has no return value.

To reopen an exposure, call the `reopenExposure` method. The method accepts three arguments.

- An exposure public ID
- An outcome enumeration from the `ExposureReopenedReason` typelist
- A `String` that describes why you are reopening the exposure

The method has no return value.

Adding to claim history from external systems

The `ClaimAPI` web service defines several methods for adding claim history events.

To add a custom history event with a specified `CustomHistoryType` but a blank description, call the `createCustomHistory` method. It takes a claim public ID, the history type (a `CustomHistoryType` enumeration). The method returns the public ID of the newly created history event.

To add a human-readable description, use the `createCustomHistoryWithDesc` method, instead. It takes an additional `String` description argument.

Creating a note from external systems

To create a note on an exposure from an external system, call the `ClaimAPI` web service method `createNote`. The method accepts a `NoteDTO` object. The argument's `ExposureID` and `ClaimID` properties identify the new note's exposure and claim IDs, respectively. The `ExposureID` setting is optional, but the `ClaimID` property must be set to a valid ID.

The method returns the public ID of the newly-created note.

Service request integration

To update service requests from vendor systems from an external system or to get information about service requests, call the `ServiceRequestAPI` web service.

Service request web service (`ServiceRequestAPI`) overview

To update service requests from Guidewire Claim Portal for Vendors or an external system or to get information about service requests, call the `ServiceRequestAPI` web service.

All methods in this web service take a vendor ID argument, which identifies the vendor contact retrieving the information or initiating the action. The vendor ID is the `AddressBookUID` property of the corresponding contact in the address book. Any client system must authenticate and confirm that the passed-in vendor ID belongs to the requesting user. The vendor ID argument is required even for methods taking other arguments, such as the service request ID, that imply a vendor ID. By requiring the vendor ID, the web service can confirm that the requested information is available to that vendor.

All method arguments have the type `String`, unless otherwise specified.

For methods that have requirements for the service request to be in a certain state, these requirements are enforced by the Gosu enhancement `GWServiceRequestStateEnhancement` in the method `createStateHandler`. That method creates new instances of a state handler object that enforces the availability of operations for the current state. You can change the logic of the state handlers or create new state handler classes.

Search for service requests

From an external system, to search for service requests for a specific vendor and some search criteria, call the `ServiceRequestAPI` web service method `searchForServiceRequests`.

Service request are returned in order of their last change date, with the most recent changes first. The maximum number of results returned is controlled by the `config.xml` parameter `ServiceRequestAPIMaxSearchResults`. If more service requests match the criteria than specified by that parameter, the server truncates results and the result object has its `ResultsAreTruncated` flag set to `true`.

The method takes the following arguments.

- Vendor ID
- Search criteria in a `ServiceRequestSearchCriteria` object. In Studio, open the `ServiceRequestSearchCriteria` Gosu class for a full list of search criteria properties.

The method returns an object of type `ServiceRequestSearchResults`, which contains only two properties.

- `ResultsAreTruncated` flag – If set to `true`, actual results exceeds the value of `config.xml` parameter `ServiceRequestAPIMaxSearchResults`.
- `Results` – A list of `ServiceRequestSummary` objects. Each object has many properties that summarize that search result. Its properties include the `ServiceRequestNumber` property for the service request number, as well as properties for the claim number, policy number, and date estimates for service. In Studio, open the Gosu class `ServiceRequestSummary` for the full list of properties.

Get service request

From an external system, to get details of a service request, call the `ServiceRequestAPI` web service method `getServiceRequest`.

The method takes the following arguments.

- Vendor ID
- Service request number, which identifies the service request. The service request must be associated with the vendor ID.

The method returns an object of type `ServiceRequestDetails`, which contains many properties including a `ServiceRequestSummary` object in the `ServiceRequestSummary` property. A `ServiceRequestSummary` has the `ServiceRequestNumber` property for the service request number, as well as properties for the claim number, policy number, and date estimates for service. In Studio, open the Gosu class `ServiceRequestSummary` for the full list of properties.

`ServiceRequestDetails` has many other properties, including vendor instructions, a quote, a list of invoices, and a list of documents. In Studio, open the Gosu class `ServiceRequestDetails` for the full list of properties.

Update service request reference number

From an external system, to update the service request reference number for a service request, call the `ServiceRequestAPI` web service method `updateServiceRequestReferenceNumber`.

It is important to understand the difference between the `ServiceRequestNumber` and `ServiceRequestReferenceNumber` types.

- `ServiceRequestNumber` is generated inside ClaimCenter and guaranteed to be unique. ClaimCenter relies on this value internally.
- `ServiceRequestReferenceNumber` is not used inside ClaimCenter, but is expected to be meaningful to the vendor. For example, it may be the ID of a related record in a third-party vendor system.

The method takes the following arguments.

- Vendor ID
- Service request number, which identifies the service request. The service request must be associated with the vendor ID.
- Service request reference number

The method has no return value.

Get document content for a service request

From an external system, to get document content for a service request, call the `ServiceRequestAPI` web service method `getDocumentContent`.

The method takes the following arguments.

- Vendor ID

- Service request number, which identifies the service request. The service request must be associated with the vendor ID.
- Document public ID

The method returns an instance of `DocumentWithContent`. It contains the following properties.

- `DocumentSummary` contains an instance of Gosu class `DocumentSummary`. The most important property in it is the `PublicID` property, which is the public ID of that document in ClaimCenter. The public ID is required when linking this document to newly-created quotes or invoices or when requesting document content.
- `DocumentContent` contains an instance of Gosu class `DocumentContent`. The document content is in the `Content` property, which contains an array of bytes. The `MimeType` property contains a MIME type.

Service request messages

A message on a service request is written by one of the parties associated with the service request. A message does not identify a recipient, but rather is posted to the service request for general viewing.

Get messages for vendor

From an external system, to get a list of all messages on active service requests associated with a specified vendor, call the `ServiceRequestAPI` web service method `getMessagesForSpecialist`.

The only method argument is a vendor ID.

The method returns a `MessagesForSpecialistResults` object with the following properties.

- The `Results` property contains a list of message objects as instances of the Gosu class `gw.webservice.cc.VERSION.vendormanagement.Message`.
- The `ResultsAreTruncated` property indicates that the list was truncated because there were too many results.

Send message to a service request

From an external system, to add a message to the service request, call the `ServiceRequestAPI` web service method `sendMessage`. The message send date is today's date. The message author is the vendor ID.

The method takes the following arguments.

- Vendor ID
- Service request number, which identifies the service request. The service request must be associated with the vendor ID.
- Message title
- Message body
- Message type, as a typecode from the typelist `ServiceRequestMessageType`. If the type of the message is `TC_QUESTION`, then ClaimCenter creates an activity to notify the user assigned to the service request.

The method has no return value.

Add or replace quote on service request

From an external system, to add an initial quote to the service request, call the `ServiceRequestAPI` web service method `addOrReplaceQuote`. If there is no quote object on the service request, this method adds the initial quote. If there is already a quote object on the service request, it is replaced with the new quote object. In both cases, the new quote becomes the active quote for the service request. The active quote object on a service request represents the aggregate of all quote information related to the service request.

The quote must have at least one linked document. This method also requires that the service request is currently in a state that supports adding a quote.

The method takes the following arguments.

- Vendor ID
- Service request number, which identifies the service request. The service request must be associated with the vendor ID.
- Creation instructions specified in an instance of the Gosu class `StatementCreationInstructions`.

The return result is the public ID of the created quote.

See also

- “Statement creation instructions reference” on page 166

Adding invoices

There are separate web service methods for adding invoices to claims or to service requests, depending on whether a service request has been created or communicated to the vendor. New invoices enter the straight-through processing flow for automatic approval and payment based on auto-processing criteria configured.

Add invoice to claim

From an external system, to add an invoice when service request does not exist or is unknown to the vendor, call the `ServiceRequestAPI` web service method `addInvoiceToClaim`. The method adds the invoice to an existing service request if one matching the input parameters is identified. Otherwise, the method creates a new service request on the claim and associates it to the incident or exposure if specified in the input parameters. You can optionally specify the incident and exposure as method arguments.

The invoice must have at least one linked document.

The method takes the following arguments.

- Vendor ID
- Claim number
- Optional incident ID, which must indicate an incident on the claim identified by the claim number
- Optional exposure ID, which must indicate an exposure on the claim identified by the claim number
- Creation instructions, as an instance of the Gosu class `StatementCreationInstructions`
- List of services performed for this invoice, as a list of `String` objects. The list is used to identify the correct service request to attach the invoice. If no matching service request is found, this method creates a new service request to perform these services.
- Creation instructions, as an instance of the Gosu class `StatementCreationInstructions`. That object includes a `LineItems` property with a list of categorized line item amounts.

The return result is the public ID of the created invoice. Save this value so that you can use it to withdraw the invoice if necessary at a later time.

Add invoice to service request

From an external system, to add an invoice to the service request, call the `ServiceRequestAPI` web service method `addInvoiceToServiceRequest`. The invoice must have at least one linked document.

The method takes the following arguments.

- Vendor ID
- Service request number, which identifies the service request. The service request must be associated with the vendor ID.
- Creation instructions, as an instance of Gosu class `StatementCreationInstructions`.

The method requires that the service request is currently in a state that supports adding an invoice.

The return value is the public ID of the created invoice. Save this value so that you can use it to withdraw the invoice if necessary at a later time.

See also

- “Statement creation instructions reference” on page 166

Update expected quote completion date for a service request

From an external system, to update the expected quote completion date to a service request, call the `ServiceRequestAPI` web service method `updateExpectedQuoteCompletionDate`.

The method requires that the service request is currently in a state that supports updating a quote exception completion date.

The method takes the following arguments.

- Vendor ID
- Service request number, which identifies the service request. The service request must be associated with the vendor ID.
- Expected completion date as a `Date` object
- Optional reason for the date change, as a `String` object. This reason is recorded in the service request history.

The method has no return value.

Update expected service completion date

From an external system, to update the expected service completion date to a service request, call the `ServiceRequestAPI` web service method `updateExpectedServiceCompletionDate`.

The method requires that the service request is currently in a state that supports updating the expected service completion date.

The method takes the following arguments.

- Vendor ID
- Service request number, which identifies the service request. The service request must be associated with the vendor ID.
- Expected completion date as a `Date` object
- Optional reason for the date change, as a `String` object. This reason is recorded in the service request history.

The method has no return value.

Add document to a service request

From an external system, to add a document to a service request, call the `ServiceRequestAPI` web service method `addDocument`.

The method takes the following arguments.

- Vendor ID
- Service request number, which identifies the service request. The service request must be associated with the vendor ID.
- Document to upload as a `DocumentContent` object. ClaimCenter will link the new document to the service request. The `DocumentContent` class has a `Content` property containing an array of bytes, and a `MimeType` property with the MIME type.

- Proposed document name. The document name is not guaranteed to be the final name for the uploaded file. For example, if the name conflicts with other document names, the final document name will change slightly.

The method returns the public ID of the created document.

Accept work for a service request

From an external system, to accept work for a service request, call the `ServiceRequestAPI` web service method `acceptWork`.

This method requires that the service request is currently in a state that supports accepting work.

The method takes the following arguments.

- Vendor ID
- Service request number, which identifies the service request. The service request must be associated with the vendor ID.
- Expected completion date (as a `Date` object) of the quotes or services, depending the type of service request.

The method has no return value.

Decline work for a service request

From an external system, to decline work for a service request, call the `ServiceRequestAPI` web service method `acceptWork`.

This method requires that the service request is currently in a state that supports declining work.

The method takes the following arguments.

- Vendor ID
- Service request number, which identifies the service request. The service request must be associated with the vendor ID.
- Optional reason for declining the work

The method has no return value.

Record work resumed for service request

From an external system, to record that work has resumed for a service request, call the `ServiceRequestAPI` web service method `recordWorkResumed`. The method is normally used when the vendor was previously prevented from making progress, but they are now able to proceed.

This method requires that the service request is currently in a state that supports resuming work.

The method takes the following arguments.

- Vendor ID
- Service request number, which identifies the service request. The service request must be associated with the vendor ID.

The method has no return value.

Record waiting for service request

From an external system, to record that the vendor is unable to proceed with the work on the service request, call the `ServiceRequestAPI` web service method `recordWaiting`. The method is normally used when the work was in progress, but is currently blocked.

The method requires that the service request is currently in a state that supports recording waiting.

The method takes the following arguments.

- Vendor ID
- Service request number, which identifies the service request. The service request must be associated with the vendor ID.
- Optional reason that the vendor is unable to proceed

The method has no return value.

Record work completed for service request

From an external system, to record that work has completed for a service request, call the `ServiceRequestAPI` web service method `recordWorkCompleted`.

The method requires that the service request is currently in a state that supports recording work completed.

The method takes the following arguments.

- Vendor ID
- Service request number, which identifies the service request. The service request must be associated with the vendor ID.

The method has no return value.

Cancel service request

From an external system, to record that the vendor does not intend to perform further work on this service request, call the `ServiceRequestAPI` web service method `cancelServiceRequest`.

The method requires that the service request is currently in a state that supports cancelling the service request. This operation can normally be performed at any time after accepting the work but before completing the work.

The method takes the following arguments.

- Vendor ID
- Service request number, which identifies the service request. The service request must be associated with the vendor ID.
- Optional reason for cancelling the service request

The method has no return value.

Withdraw invoice for service request

From an external system, to record that the vendor wants to withdraw an invoice for a service request, call the `ServiceRequestAPI` web service method `withdrawInvoice`.

The method requires that the service request is currently in a state that supports withdrawing an invoice. This operation can normally be performed at any time before it is paid or rejected.

The method takes the following arguments.

- Vendor ID
- Service request number, which identifies the service request. The service request must be associated with the vendor ID.
- Invoice public ID of the invoice to withdraw. Typically this would be obtained as the return result of the `addInvoiceToServiceRequest` method.

- Optional reason for withdrawing the invoice

The method has no return value.

Statement creation instructions reference

Several `ServiceRequestAPI` methods use the `StatementCreationInstructions` object. The `StatementCreationInstructions` object contains many types of data about an invoice or a quote. The object includes the following properties.

- `Currency` – The invoice currency
- `StatementCreationTime` – The date the statement was created
- `LineItems` – Array of categorized amounts, each one of which is an instance of the type `ServiceRequestStatementLineItem`.
- `Description` – Description
- `QuoteNumberOfDaysToPerformService` – For quotes, the number of days the vendor expects to require to perform the service
- `ExistingDocumentsToLinkPublicIds` – A list of public IDs of `Document` entity instances to link to the new quote or invoice. The documents must already be linked to this service request. To get the list of documents for this service request, get the list of `Documents` on the associated `ServiceRequestDetails` object for this service request.
- `DocumentsToUpload` – A list of `DocumentContent` objects containing the contents of additional documents to create and then link to the new quote or invoice. Each object has a `Content` property containing an array of bytes, and `MimeType` property with the MIME type. Also see property `DocumentsToUploadNames`.
- `DocumentsToUploadNames` – A list of names for the documents passed in the `DocumentsToUpload` property. This list must have the same number of items as `DocumentsToUpload`.
- `ReferenceNumber` – An optional number to assign to the created quote or invoice. This number is not used inside ClaimCenter but is expected to be meaningful to the vendor. For example, it may be the ID of a related record in a third-party system.
- `Source` – An optional field to capture where the quote or invoice came from, such as manual entry or through a portal. The type is a typecode from the `StatementSource` typelist.

part 3

Plugins

ClaimCenter plugins are classes that ClaimCenter invokes to perform an action or calculate a result at a specific time in its business logic. ClaimCenter defines plugin interfaces for various purposes. You can write your own implementations of plugins in Gosu or Java.

- Perform calculations.
- Provide configuration points for your own business logic at clearly defined places in application operation, such as validation or assignment.
- Generate new data for other application logic, such as generating a new claim number.
- Define how the application interacts with other Guidewire InsuranceSuite applications for important actions relating to claims, policies, billing, and contacts.
- Define how the application interacts with other third-party external systems such as a document management system, third-party claim systems, third-party policy systems, or third-party billing systems.
- Define how ClaimCenter sends messages to external systems.

You can implement a plugin in the programming languages Gosu or Java. In many cases, it is easier to implement a plugin with a Gosu class. If you use Java, you must use a separate IDE other than Studio, and you must regenerate Java API libraries after any data model changes.

If you implement a plugin in Java, optionally you can write your code as an OSGi bundle. The OSGi framework is a Java module system and service platform that helps cleanly isolate code modules and any necessary Java API libraries. Guidewire recommends OSGi for all new Java plugin development.

Overview of plugins

ClaimCenter defines a plugin as an *interface*, which is a set of methods that are necessary for a specific task. Each plugin interface is a strict contract of interaction and expectation between the application and the plugin implementation. Some other set of code that implements the interface must perform the task and return any appropriate result values.

Conceptually, there are two main steps to implement a plugin.

1. Write a Gosu or Java class that implements a plugin interface
2. Register the plugin implementation class

For most plugin interfaces, you can register only a single plugin implementation for that interface. However, some plugin interfaces support multiple implementations for the same interface, such as messaging plugins and startable plugins.

The plugin itself might do most of the work or it might interact with other external systems. Many plugins run while users wait for responses from the application user interface. Guidewire strongly recommends that you carefully consider response time, including network response time, as you write your plugin implementations.

Implementing plugin interfaces

Choose a plugin implementation type

There are several ways to implement a ClaimCenter plugin interface.

- Implement a Gosu class. In many cases it is easiest to implement a plugin interface as a Gosu class because you write and debug the plugin code in Guidewire Studio. The Gosu language has powerful features that simplify the creation of a plugin class, including type inference and easy access to web services and XML.
- Implement a Java class. A Java plugin class must be implemented in an IDE other than Studio. Any Java IDE can be used. You can choose to use the included application called IntelliJ IDEA with OSGi Editor for your Java plugin development even if you do not choose to use OSGi. Also, if you write your plugin in Java, you must regularly regenerate the Java API libraries after changes to data model configuration.
- Implement a Java class encapsulated in an OSGi bundle. The OSGi framework is a Java module system and service platform that helps cleanly isolate code modules and any necessary Java API libraries. Guidewire recommends OSGi for all new Java plugin development. To simplify OSGi configuration, ClaimCenter includes an application called IntelliJ IDEA with OSGi Editor.

The following table compares the types of plugin implementations.

Features of each plugin implementation type	Gosu plugin (no OSGi)	Java plugin	OSGi plugin (Java with OSGi)
Choice of development environment			
You can use Guidewire Studio to write and debug code.	•		
You can use the included application IntelliJ IDEA with OSGi editor to write code.		•	•
Usability			
Native access to Gosu blocks, collection enhancements, Gosu classes.	•		
Entity and typecode APIs are the same as for Rules code and PCF code.	•		
Requires regenerating Java API libraries after data model changes.		•	•
Third-party Java libraries			
Your plugin code can use third-party Java libraries.	•	•	•
You can embed third-party Java libraries within an OSGi bundle to reduce conflicts with other plugin code or ClaimCenter itself.			•
Dependencies on specific third-party packages and classes are explicit in manifest files and validated at startup time.			•

Writing a plugin implementation class

Store plugin classes in a package located in your own hierarchy, such as `mycompany.plugins`. Never store a class in the internal `gw.*` or `com.guidewire.*` package hierarchies.

Create a class that implements the plugin's interface.

```
public class SamplePluginClass implements sampleInterface
```

Implement all public methods of the interface. Guidewire Studio and most professional Java IDEs can detect when required methods are missing from a class. Studio provides a tool that can create stub versions of unimplemented methods.

The `@PluginParameter` annotation

Plugin classes can describe their parameters with the `@PluginParameter` annotation. The annotation can specify various properties of the parameter, such as its type and whether it is required.

@PluginParameter Description	
helptext	Debug message information. Default is "".
name	Parameter name. To enable the name to be accessible to external tools without loading the class file, make the property value a string literal or regular expression.
required	Boolean value specifying whether the parameter is required. Default is <code>false</code> .
type	Type of parameter. Type validation is performed on the parameter. Supported parameter types are listed below. Default is <code>String</code> . <ul style="list-style-type: none"> • <code>Boolean</code> • <code>Date</code> • <code>EmailAddress</code> • <code>File</code> • <code>Integer</code>

@PluginParameter Description

- String
- StringArray
- URL

The following Gosu code segment demonstrates how to use the @PluginParameter annotation.

```
@PluginParameter(:name="identifier", :type=Integer, :required=true)
@PluginParameter(:name="key", :type=String)
@PluginParameter(:name="iterationCount", :type=Integer)
class samplePluginClass implements sampleInterface, InitializablePlugin {
    // Define a string constant for each parameter name
    public final static var ID_PARAM : String = "identifier"
    public final static var KEY_PARAM : String = "key"
    public final static var ITERATION_PARAM : String = "iterationCount"

    // Class parameter variables
    var _identifier : Integer
    var _key : String
    var _iterationCount : Integer

    // Demonstrate various techniques to access parameter values
    override function setParameters(params : Map<Object, Object>) {
        _identifier = params.get(ID_PARAM) as Integer
        _key = getParameter(params, KEY_PARAM, "Sample debug message").toCharArray()
        _iterationCount = getIntegerParameter(params, ITERATION_PARAM, 20)

        // ... Remainder of class implementation ...
    }
}
```

General plugin behavior

Plugin methods must strive to be idempotent so that calls to the method during a single transaction produce the same results whether executed once or multiple times. Plugin methods must also attempt to be reentrant.

If the interface contains a method starting with the substring `get`, `set`, or `is` and takes no parameters, define the method using the property getter or setter syntax. Do not implement the method using the name as it is defined in the interface. For example, if `samplePluginClass` declares the method `getMyVar()`, the implementation must define a property getter method, as demonstrated in the following code segment.

```
class samplePluginClass implements sampleInterface {

    property get MyVar() : String {
        // ... Property getter implementation ...
    }
}
```

For plugins written in Java, the Java API libraries must be regenerated whenever changes are performed on the data model configuration.

Base configuration plugin implementation classes

For most plugin interfaces, a base configuration class implementation is provided and registered in the Studio plugin registry. A few of these base implementations can be used as-is in a production environment. However, base implementations are typically provided for demonstration purposes only. Their intention is to be used as a beginning foundation to which extra configuration code is added before being placed in a production environment.

Some plugin implementations connect with other Guidewire InsuranceSuite applications. The InsuranceSuite plugin implementations are stored in packages with names that reference the target application and its version number. For example, a package name for a plugin that integrates with ClaimCenter 9.0.0 would include the characters `cc900`. This naming convention assists in achieving a smooth migration and backward compatibility between Guidewire applications.

Plugin implementation classes in the base configuration

For some plugin interfaces, the base configuration of ClaimCenter provides a plugin implementation that you can use instead of writing your own version. Some plugin implementation classes are preregistered in the base configuration.

Some plugin implementations are for demonstration only. Check the documentation for each plugin interface or contact Customer Support if you are not sure whether an included plugin implementation is supported for production use.

ClaimCenter includes plugin implementations to connect to other Guidewire applications in Guidewire InsuranceSuite. The plugin implementations for InsuranceSuite integration are located in packages that include the intended target application and version number. This package structure helps ensure smooth migration and backwards compatibility among Guidewire applications. Carefully confirm package names for any plugin implementations that you want to use. The package name typically includes the Guidewire application two-character abbreviation, followed by the application release number with no periods. For example, the name of a package that supports ClaimCenter 10.0.0 includes cc1000. Be sure to use the plugin implementation class that matches the release of the other application, not the current application.

Registering a plugin implementation class

After implementing the plugin class, the class must be registered so ClaimCenter knows about it. The registry configures which implementation class is responsible for which plugin interface. If you correctly register your plugin implementation, ClaimCenter calls the plugin at the appropriate times in the application logic. The plugin implementation performs some action or computation and in some cases returns results back to ClaimCenter.

To register a plugin, in Studio in the **Project** window, navigate to **configuration > config > Plugins > registry**. Right-click on **registry**, and choose **New > Plugin**.

To use the Plugins registry, you must know all of the following.

- The plugin interface name. You must choose a supported ClaimCenter plugin interface. You cannot create your own plugin interface.
- The fully-qualified class name of your concrete plugin implementation class.
- Any initialization parameters, also called plugin parameters.

In nearly all cases, you must have only one active enabled implementation for each plugin interface. However, there are exceptions, such as messaging plugins, encryption plugins, and startable plugins. If the plugin interface supports only one implementation, be sure to remove any existing registry entries before adding new ones.

As you register plugins in Studio, the interface prompts you for a plugin name. Plugin names can include alphanumeric characters only. Space or blank characters are not allowed.

If the interface accepts only one implementation, the plugin name is arbitrary. However, it is the best practice to set the plugin name to match the plugin interface name and omit the package.

If the interface accepts more than one implementation, the plugin name may be important. For example, for messaging plugins, enter the plugin name when you configure the messaging destination in the separate Messaging editor in Studio. For encryption plugins, if you ever change your encryption algorithm, the plugin name is the unique identifier for each encryption algorithm.

If the interface field is blank, ClaimCenter assumes the interface name matches the plugin name. You might notice that some of the plugin implementations that are pre-registered in the default configuration have that field blank for this reason.

The **Environment** field can accept multiple values specified in a comma-separated list.

Plugin parameters

In the Plugins Registry editor, you can add a list of parameters as name-value pairs. The plugin implementation can use these values. For example, you might pass a server name, a port number, or other configuration information to your plugin implementation code using a parameter. Using a plugin parameter in many cases is an alternative to using hard-coded values in implementation code.

You can use external server configuration to use placeholders to set the parameter values dynamically rather than using static values. By using this technique, you can modify the parameter values without rebuilding and redeploying a WAR or EAR file for ClaimCenter.

To use plugin parameters, a plugin implementation must implement the `InitializablePlugin` interface in addition to the main plugin interface. If ClaimCenter detects that your plugin implementation implements `InitializablePlugin`, ClaimCenter calls your plugin's `setParameters` method. That method must have exactly one argument, which is a `java.util.Map` object. In the map, the parameters names are keys in the map.

By default, all plugin parameters have the same name-value pairs for all values of the servers and environment system variables. However, the Plugins registry allows optional configuration by server, by environment, or both. For example, you could set a server name differently depending on whether you are running a development or production configuration.

See also

- “Getting plugin parameters from the plugins registry editor” on page 175
- *Administration Guide*

For Java plugins (without OSGi), define a plugin directory

For Java plugin implementations that do not use OSGi, the Plugins Registry editor has a field for a plugin directory. A plugin directory is where you put non-OSGi Java classes and library files. In this field, enter the name of a subdirectory in the `ClaimCenter/modules/configuration/plugins` directory.

To reduce the chance of conflicts in Java classes and libraries between plugin implementations, define a unique name for a plugin directory for each plugin implementation. If you do not specify a plugin directory, the default is shared.

OSGi plugin implementations automatically isolate code for your plugin with any necessary third-party libraries in one OSGi bundle. Therefore, for OSGi plugin implementations, you do not configure a plugin directory in the Plugins Registry editor in Studio.

Deploying Java files, including Java code called from Gosu plugins

To deploy any Java files or third-party Java libraries varies based on your plugin type, perform the following operations.

- If your Gosu plugin implements the plugin interface but accesses third-party Java classes or libraries, you must put these files in the right places in the configuration environment. It is important to note that the plugin directory setting discussed in that section has the value `Gosu` for code called from Gosu.
- For Java plugins that do not use OSGi, first ensure you define a plugin directory.
- For Java plugin classes deployed as OSGi bundles, deployment of your plugin files and third party libraries is very different from deploying non-OSGi Java code. ClaimCenter supports OSGi bundles only to implement a ClaimCenter plugin interface and any of your related third-party libraries. It is unsupported to install OSGi bundles for any other purpose.

See also

- “Deploying non-OSGi Java classes and JAR files” on page 54
- “Using IntelliJ IDEA with OSGi editor to deploy an OSGi plugin” on page 56

Error handling in plugins

ClaimCenter tries to respond appropriately to errors that occur during the execution of a plugin method. When possible, a default action is performed. However, when an error occurs in certain cases, such as the generation of a policy or claim number, no reasonable default behavior exists. In such cases, the action that triggered the plugin fails and displays an error message.

Plugin error handling is dependent on the plugin interface and program context. Check the plugin method signature to view the possible exceptions.

A plugin method can throw custom exceptions. The custom exception can be any of the following types.

- A Gosu-accessible exception declared in the `gw.api.util` package
- A Gosu-accessible exception declared in the plugin's module
- A Java built-in exception, such as `IllegalArgumentException` or `IOException`

The `DisplayableException` shows an error message in the user interface.

```
uses gw.api.util.DisplayableException

public class MyCustomPluginHandler {
    function myPluginHandler(){
        throw new DisplayableException("MyCustomPluginHandler: myPluginHandler Error Message")
    }
}
```

Enabling or disabling a plugin

Navigate to the Plugins Registry editor and select the relevant plugin. Clear or select the **Disabled** check box to enable or disable the plugin.

You can use external server configuration to use a placeholder to set the value of this flag dynamically. By using this technique, you can enable or disable the plugin without rebuilding and redeploying a WAR or EAR file for ClaimCenter. To set an external configuration placeholder as the value for the `disabled` attribute of the plugin, you must use the **Text** tab in the plugin editor.

If you have more than one class that implements a plugin, you must ensure than only one class is enabled. If you use the **Environment** property to determine which class ClaimCenter uses, do not include the same environment in that property for more than one class. ClaimCenter does not support multiple implementation classes for the same environment, even if only one is enabled. Similarly, do not leave this property blank for more than one class.

See also

- *Administration Guide*

Example Gosu plugin

The following example code implements a simple Gosu messaging plugin that uses parameters.

```
uses java.util.Map
uses java.plugin

class MyTransport implements MessageTransport, InitializablePlugin {

    private var _servername : String
    private var _destinationID : int

    // Note the empty constructor. If you provide an empty constructor, the application calls it
    // as the plugin instantiates, which occurs before the application calls setParameters().
    construct() {}

    override function setParameters(parameters : Map<String,String>) {
        // Access values in the Map argument to retrieve parameters defined in Studio Plugins registry
        _servername = parameters["MyServerName"] as String
    }

    override function suspend() {}

    override function shutdown() {}

    override function setDestinationID(id:int) { _destinationID = id }

    override function resume() {}

    override function send(message : entity.Message, transformedPayload : String) {
        print("MESSAGE SEND ===== ${message.Payload} --> ${transformedPayload}")
        message.reportAck()
    }
}
```

Special notes for Java plugins

Users can trigger a plugin call by performing a user-interface action. External applications can trigger a plugin by calling a web service. It can sometimes be useful for a Java plugin method to know which user or application triggered the plugin call.

The `CurrentUserUtil` utility class and its `getCurrentUser` method can be called by a Java plugin to retrieve the triggering user or application. Sample code is shown below. The `getUser` method returns a `User` entity.

```
trigger = CurrentUserUtil.getCurrentUser().getUser();
```

See also

- ““Access Entity Data in Java””.

Getting plugin parameters from the plugins registry editor

In the Studio Plugins Registry editor, you can add one or more optional parameters to pass to your plugin during initialization. For example, you could use the editor to pass server names, port numbers, timeout values, or other settings to your plugin code. The parameters are pairs of `String` values, also known as name/value pairs. ClaimCenter treats all plugin parameters as text values, even if they represent numbers or other objects.

To use the plugin parameters in your plugin implementation, your plugin must implement the `InitializablePlugin` interface in addition to the main plugin interface.

If you do this, ClaimCenter calls your plugin’s `setParameters` method. That method must have exactly one argument, which is a `java.util.Map` object. In the map, the parameters names are keys in the map, and they map to the values from Studio.

See also

- For an implementation of a messaging plugin that uses parameters, see “Example Gosu plugin” on page 174.

Getting the local file path of the root and temp directories

For Gosu and Java plugins, the names of the plugin’s root and temporary directory paths can be retrieved by accessing built-in properties from the `Map`. These properties are not available from OSGi plugins.

The root directory name is stored in the static variable `InitializablePlugin.ROOT_DIR`.

The temporary directory name is stored in the static variable `InitializablePlugin.TEMP_DIR`.

Plugin registry APIs

The `gw.plugin` package contains the `Plugins` class that provides static utility methods for accessing plugins in the registry and checking whether a plugin is enabled. You can use these methods from both Java and Gosu.

This functionality is useful if you want to access one plugin from another plugin. For example, a messaging-related plugin that you write might need a reference to another messaging-related plugin to communicate or share common code.

You can also use these methods to access plugin interfaces that provide services to plugins or other Java code.

For example, the `IScriptHost` plugin can evaluate Gosu expressions. To use it, get a reference to the currently-installed `IScriptHost` plugin. Next, call its methods. Call the `putSymbol` method to make a Gosu context symbol, such claim to evaluate to a specific `Claim` object reference. Call the `evaluate` method to evaluate a `String` containing Gosu code.

Getting references to plugins

To access the currently implemented instance of a plugin, call the `Plugins.get` static method and pass the plugin interface name as an argument. This method returns a reference to the plugin implementation. In Gosu, the return result

is properly statically typed so you can directly call methods on the result. The following code block demonstrates this process.

```
var plugin = gw.plugin.Plugins.get(ContactSystemPlugin)
try {
    plugin.retrieveContact("abc:123")
} catch(e){
    throw new DisplayableException(e.Message)
}
```

Alternatively, you can request a plugin by the plugin name defined in the Plugins registry. This syntax is used when there is more than one plugin implementation of the interface. For example, multiple implementations of an interface are common in messaging plugins. In such cases, use the `get` method signature that takes a `String` for the plugin name.

```
var plugin = gw.plugin.Plugins.get("MyContactPluginRegistryName") as ContactSystemPlugin
```

Checking whether a plugin is enabled

You can call the `isEnabled` method of the `Plugins` class to determine if a plugin is enabled in Studio. Pass either of the following `String` arguments.

- The interface name type with no package
- The plugin name defined in the Plugins registry

The following code segment demonstrates how to call the `isEnabled` method.

```
var contactSystemEnabled = gw.plugin.Plugins.isEnabled(ContactSystemPlugin)
```

Plugin thread safety

If you register a Java plugin or a Gosu plugin, exactly one instance of that plugin exists in the Java virtual machine on that server, generally speaking. For example, if you register a document production plugin, exactly one instance of that plugin instantiates on each server.

The rules are different for messaging plugins in ClaimCenter server clusters. In general, messaging plugins instantiate on servers with the messaging server role. Servers without that role typically have no instances of message request, message transport, and message reply plugins. Messaging plugins must be especially careful about thread safety because messaging supports a large number of simultaneous threads, configured in Studio.

However, one server instance of the Java plugin or Gosu plugin must service multiple user sessions. Because multiple user sessions use multiple process threads, follow the these rules to avoid thread problems.

- Your plugin must support multiple simultaneous calls to the same plugin method. A plugin can be called from either ClaimCenter or from different threads. You must ensure multiple calls to the plugin never access the same shared data. Alternatively, protect access to shared resources so that two threads do not access it simultaneously.
- Your plugin implementation must support multiple user sessions. Generally speaking, do not assume shared data or temporary storage is unique to one user request (one HTTP request of a single user).

The most important way to avoid thread safety problems in plugin implementations is to avoid variables stored once per class, referred to as static variables. Static variables are a feature of both the Java language and the Gosu language. Static variables let a class store a value once per class, initialized only once. In contrast, object instance variables exist once per instance of the class.

Static variables can be extremely dangerous in a multi-threaded environment. Using static variables in a plugin can cause serious problems in a production deployment without taking great care to avoid problems. Be aware that such problems, if they occur, are extremely difficult to diagnose and debug. Timing in an multi-user multi-threaded environment is difficult, if not impossible, to control in a testing environment.

Because plugins could be called from multiple threads, there is sometimes no obvious place to store temporary data that stores state information. Where possible and appropriate, replace static variables with other mechanisms, such as

setting properties on the relevant data passed as parameters. For example, in some cases perhaps use a data model extension property on a `Claim` or other relevant entity (including custom entities) to store state-specific data for the plugin. Be aware that storing data in an entity shares the data across servers in a ClaimCenter cluster. Additionally, even standard instance variables (not just static variables) can be dangerous because there is only one instance of the plugin.

If you are experienced with multi-threaded programming and you are certain that static variables are necessary, you must ensure that you synchronize access to static variables. Synchronization refers to a feature of Java that locks access between threads to shared resources, such as static variables.

Using Java concurrent data types, even from Gosu

The simplest way of synchronizing access to a static variable in Java is to store data as an instance of a Java classes defined in the package `java.util.concurrent`. The objects in that package automatically implement synchronization of their data, and no additional code or syntax is necessary to keep all access to this data thread-safe. For example, to store a mapping between keys and values, instead of using a standard Java `HashMap` object, instead use `java.util.concurrent.ConcurrentHashMap`.

These tools protect the integrity of the keys and values in the map. However, you must ensure that if multiple threads or user sessions use the plugin, the business logic still does something appropriate with shared data. You must test the logic under multi-user and multi-thread situations.

Be aware that all thread safety APIs that use blocking can affect performance negatively. For high performance, use such APIs carefully and test all code under heavy loads that test the concurrency.

Using synchronized methods (Java only)

Java provides a feature called synchronization that protects shared access to static variables. It lets you tag some or all methods so that no more than one of these methods can be run at once. Then, you can add code safely to these methods that get or set the object's static class variables, and such access are thread safe.

If an object is visible to more than one thread, and one thread is running a synchronized method, the object is locked. If an object is locked, other threads cannot run a synchronized method of that object until the lock releases. If a second thread starts a synchronized method before the original thread finishes running a synchronized method on the same object, the second thread waits until the first thread finishes. This is known as blocking or suspending execution until the original thread is done with the object.

Mark one or more methods with this special status by applying the `synchronized` keyword in the method definition. This example shows a simple class with two synchronized methods that use a static class variable.

```
public class SyncExample {  
    private static int contents;  
  
    public int get() {  
        return contents;  
    }  
  
    // Define a synchronized method. Only one thread can run a syncced method at one time for this object  
    public synchronized void put1(int value) {  
        contents = value;  
        // do some other action here perhaps...  
    }  
  
    // Define a synchronized method. Only one thread can run a syncced method at one time for this object  
    public synchronized void put2(int value) {  
        contents = value;  
        // Do some other action here perhaps...  
    }  
}
```

Synchronization protects invocations of all synchronized methods on the object; it is not possible for invocations of two different synchronized methods on the same object to interleave. For the earlier example, the Java virtual machine does all of the following.

- Prevents two threads simultaneously running `put1` at the same time

- Prevents `put1` from running while `put2` is still running
- Prevents `put2` from running while `put1` is still running.

This approach protects integrity of access to the shared data. However, you must still ensure that if multiple threads or user sessions use the plugin, your code does something appropriate with this shared data. Always test your business logic under multi-user and multi-thread situations.

ClaimCenter calls the plugin method initialization method `setParameters` exactly once, hence only by one thread, so that method is automatically safe. The `setParameters` method is a special method that ClaimCenter calls during plugin initialization. This method takes a Map with initialization parameters that you specify in the Plugins registry in Studio.

On a related note, Java class constructors cannot be synchronized; using the Java keyword `synchronized` with a constructor generates a syntax error. Synchronizing constructors does not make sense because only the thread that creates an object has access to it during the time Java is constructing it.

Using Java synchronized blocks of code (Java only)

Java code can also synchronize access to shared resources by defining a block of statements that can only be run by one thread at a time. If a second thread starts that block of code, it waits until the first thread is done before continuing. Compared to the method locking approach described earlier in this section, synchronizing a block of statements allows much smaller granularity for locking.

To synchronize a block of statements, use the `synchronized` keyword and pass it a Java object or class identifier. In the context of protecting access to static variables, always pass the class identifier `ClassName.class` for the class hosting the static variables.

For example, the following code segment demonstrates statement-level or block-level synchronization.

```
class MyPluginClass implements IMyPluginInterface {
    private static byte[] myLock = new byte[0];
    public void MyMethod(Address f){
        // SYNCHRONIZE ACCESS TO SHARED DATA!
        synchronized(MyPluginClass.class){
            // Code to lock is here....
        }
    }
}
```

This finer granularity of locking reduces the frequency that one thread is waiting for another to complete some action. Depending on the type of code and real-world use cases, this finer granularity could improve performance greatly over using synchronized methods. This is particularly the case if there are many threads. However, you might be able to refactor your code to convert blocks of synchronized statements into separate synchronized methods.

Both approaches protect integrity of access to the shared data. However, you must plan to handle multiple threads or user sessions to use your plugin, and do safely access any shared data. Also, test your business logic under realistic heavy loads for multi-user and multi-thread situations.

Avoiding singletons because of thread-safety issues

Thread safety problems apply to any Java object that has only a single instance—also referred to as a singleton—implemented using static variables. Because accessing static variables in multi-threaded code is complex, Guidewire strongly discourages using singleton Java classes. You must synchronize access to all data singleton instances just as for other static variables as described earlier in this section. This restriction is important for all Gosu Java that ClaimCenter runs.

The following code segment provides an example of creating a singleton using a class static variable.

```
public class MySingleton {
    private static MySingleton _instance = new MySingleton();
    private MySingleton() {
        // Construct object . . .
    }
}
```

```
    }
    public static MySingleton getInstance() {
        return _instance;
    }
}
```

Design plugin implementations to support server clusters

Generally speaking, if your plugin deploys in a ClaimCenter server cluster, there are instances of the plugin deployed on every server in the cluster. Consequently, design your plugin code (and any associated integration code) to support concurrent instances. If the Gosu code calls out to Java for any network connections, that code must support concurrent connections.

There is an exception for this cluster rule: In general, messaging plugins instantiate only on servers with the `messaging` server role.

Because there may be multiple instances of the plugin, you must ensure that you update a database from Java code carefully. Your code must be thread safe, handle errors fully, and operate logically for database transactions in interactions with external systems. For example, if several updates to a database must be treated as one action or several pieces of data must be modified as one atomic action, design your code accordingly.

General plugin thread safety synchronization techniques are insufficient to synchronize data shared across multiple servers in a cluster. Additional safeguards are required because each server has its own Java virtual machine and, therefore, its own data space. You must implement your own approach to ensure access to shared resources safely even if accessed simultaneously by multiple threads and on multiple servers. Write your plugins to know about the other server's plugins but not to rely on anything other than the database to communicate among each other across servers.

Reading system properties in plugins

You can test plugins in multiple deployment environments without recompiling plugins. For example, if a plugin runs on a test server, then the plugin queries a test database. If the plugin runs on a production server, then the plugin queries a production database.

Alternatively, you might want a plugin that can be run on multiple machines in a cluster with each machine having its own identity. You can implement unique behavior within the cluster. Additionally, you can add this information to log files both on the local machine and in the external system.

In these cases, plugins can use environment (`env`) and server ID (`serverid`) deployment properties to deploy a single plugin with different behaviors in multiple contexts or across clustered servers. Define these system properties in the server configuration file or as command-prompt parameters for the command that launches your web server container. In general, use the default system property settings. If you want to customize them, use the Plugin Registry editor in Guidewire Studio™.

Gosu plugins can query system properties by using the following `gw.api.system.server.ServerUtil` static methods:

getEnv

Get runtime values for the environment for this cluster member.

getServerId

Get the runtime value for the server ID for this cluster member.

getServerRoles

Get the set of server roles for this cluster member.

To query general system properties, use the `getProperty` static method of `java.lang.System` and pass a property name. For example:

```
java.lang.System.getProperty("gw.cc.env");
```

Do not call local web services from plugins

Do not call locally-hosted SOAP web service APIs from within a plugin or the rules engine in production systems. If the web service hosted locally modifies entity data and commits the bundle, the current transaction does not always detect and reload local data for your plugin implementation. Instead, refactor your code to avoid this case. For example, write a Gosu class that performs a similar function as the web service but that does not commit the bundle. This type of refactoring also results in higher server performance.

Creating unique numbers in a sequence

Typical ClaimCenter implementations need to reliably create unique numbers in a sequence for some types of objects. For example, to enforce a series of unique IDs, such as public ID values, within a sequence. You can generate new numbers in a sequence using sequence generator APIs in the `SequenceUtil` class.

These methods take two parameters.

- An initial value for the sequence, if it does not yet exist.
- A `String` with up to 26 characters that uniquely identifies the sequence. This is the sequence key (`sequenceKey`).

For example, the following Gosu code gets a new number from a sequence called `WidgetNumber`.

```
nextNum = gw.api.system.database.SequenceUtil.next(10, "WidgetNumber")
```

If this is the first time any code has requested a number in this sequence, the value is 10. If other code calls this method again, the return value is 11, 12, and so on, depending on the number of times code has requested numbers for this sequence key.

You can disable the sequence utility class by setting the `config.xml` parameter `DisableSequenceUtil`. Use this to ensure that any sequences in your code use some alternative mechanism for sequenced identifiers.

Restarting and testing tips for plugin developers

If you frequently modify your plugin code, you might need to frequently redeploy ClaimCenter to test your plugins. If it is a non-production server, you may not want to shut down the entire web application container and restart it. For development use only, reload only the ClaimCenter application rather than the web application container. If your web application container supports this, replace your plugin class files and reload the application.

Summary of ClaimCenter plugins

The following tables summarize the plugin interfaces that ClaimCenter defines.

Plugins for general purpose use

Plugin Interface	Description
<code>BackgroundTaskLoadBalancingPlugin</code>	Defines strategies for managing the load balancing of messaging destinations and startable services.
<code>ClusterBroadcastTransportFactory</code>	Provides support for communication among nodes in a Guidewire ClaimCenter cluster. See <i>Administration Guide</i> for more information.
<code>ClusterUnicastTransportFactory</code>	
<code>RedisBroadcastTransportFactory</code>	
<code>CredentialsPlugin</code>	Enables user authentication to an external system by retrieving the user name and password required to gain access to the system. See “ Credentials plugin ” on page 283.

Plugin Interface	Description
IActivityEscalationPlugin	Overrides the behavior of activity escalation instead of simply calling rule sets. See “Exception and escalation plugins” on page 287.
IAddressAutocompletePlugin	Configures how address automatic completion and fill-in operate. See “Automatic address completion and fill-in plugin” on page 293.
IBaseUrlBuilder	Generates a base URL to use for web application pages affiliated with this application, given the HTTP servlet request URI (<code>HttpServletRequest</code>). See “Defining base URLs for fully qualified domain names” on page 291.
IBatchCompletedNotification	Launches custom actions after a work queue or batch process completes processing a batch of items.
IEmailTemplateSource	Retrieves email templates. In the base configuration, the templates are retrieved from the server file system, but the plugin can be customized to access an alternative location.
IEncryptionPlugin	<p>Encodes or decodes a <code>String</code> based on an algorithm you provide to hide important data, such as bank account numbers or private personal data. ClaimCenter does not provide any encryption algorithm in the product. ClaimCenter simply calls this plugin implementation, which is responsible for encoding an unencrypted <code>String</code> or reversing that process. The implementation of this plugin in the base configuration does nothing. See “Encryption integration” on page 263.</p> <p>You can register multiple implementation for this interface.</p>
IGroupExceptionPlugin	Overrides the behavior of group exceptions instead of simply calling rule sets. See “Exception and escalation plugins” on page 287.
InboundIntegrationStartablePlugin	<p>High performance inbound integrations, with support for multi-threaded processing of work items. See “Multi-threaded inbound integration” on page 307.</p> <p>You can register multiple implementation for this interface to communicate with multiple external systems.</p>
INoteTemplateSource	Retrieves note templates. In the base configuration, the templates are retrieved from the server file system, but the plugin can be customized to retrieve them from a document management system.
IPhoneNormalizerPlugin	Normalizes phone numbers that users enter through the application and that enter the database through data import. See “Phone number normalizer plugin” on page 293.
IPreupdateHandler	Implements your preupdate handling in plugin code rather than in rules. See “Preupdate handler plugin” on page 284.
IProcessesPlugin	Instantiates custom batch processing classes so they can be run on a schedule or on demand. See “Implementing IProcessesPlugin” on page 736.
IRestDispatchPlugin	<p>Processes REST API requests, including the following:</p> <ul style="list-style-type: none"> • Preprocess incoming REST API requests • Rewrite outgoing API responses • Control how and what ClaimCenter logs for each API request
IScriptHost	Evaluates Gosu code to provide context-sensitive data to other services. For example, this service could evaluate a <code>String</code> that has the Gosu expression <code>"policy.myFieldName"</code> at run time.
IStartablePlugin	Creates new plugins that immediately instantiate and run on server startup. See “Startable plugins” on page 299.

Plugin Interface	Description
	You can register multiple implementation for this interface.
ITestingClock	Used for testing complex behavior over a long span of time, such as multiple billing cycles or timeouts that are multiple days or weeks later. This plugin is for development (non-production) use only. It programmatically changes the system time to accelerate passing of time in ClaimCenter.
	See “Testing clock plugin (for non-production servers only)” on page 295.
IUserExceptionPlugin	Overrides the behavior of user exceptions instead of simply calling rule sets. See “Exception and escalation plugins” on page 287.
IWorkItemPriorityPlugin	Calculates a the processing priority of a work item. See “Work item priority plugin” on page 287.
ManagementPlugin	The external management interface for ClaimCenter, which enables you to implement management systems, such as JMX and SNMP. See “Management integration” on page 277.

Plugins specific to Guidewire ClaimCenter

Plugin Interface	Description
IAggregateLimitTransactionPlugin	Provides an optional capability to determine if a Transaction applies to an Aggregate Limit, beyond the usual values on the Aggregate Limit screen.
IApprovalAdapter	(1) Determines whether a user has any authority to submit something, typically a financial transaction. If so, determines whether the user has sufficient authority or requires further approval. If no authority, then rejects the request. (2) Determines who needs to give approval next. In general do not implement this plugin because approval rule sets are sufficient for most cases. By default, ClaimCenter uses internal logic to compare financial totals to authority limits. Also, by default uses the approval rule sets to determine the approving person. See “Approval plugin” on page 293.
IArchivingSource	Stores and retrieves archived claims from an external backing source. See “Archiving integration” on page 653.
IClaimNumGenAdapter	Generates unique claim numbers for new claims. Generates temporary claim numbers for draft claims. You must implement this plugin. However, ClaimCenter includes a demo version of this plugin. See “Claim number generator plugin” on page 292.
IGeocodePlugin	Geocoding support in ClaimCenter and ContactManager. See “Geographic data integration” on page 249.
IPolicyRefreshPlugin	Supports policy refresh. See “Policy refresh overview” on page 497.
IPolicySearchAdapter	(1) Searches policies to find likely ones to link to a claim (2) Retrieves full policy information for the selected policy. Used during the process of setting up a new claim. Searches can return zero rows if there is no integration to a policy system. You must implement this plugin. However, ClaimCenter includes a demo version of this plugin. See “Policy search plugin” on page 472.
IReinsurancePlugin	Retrieves reinsurance information from a policy administration system. See “Reinsurance integration” on page 261.
PolicyLocationSearchPlugin	Performs a policy location search in a policy system such as PolicyCenter. See “Policy location search plugin” on page 496.

Plugins for ClaimCenter financials

Plugin Interface	Description
IBackupWithholdingPlugin	Handles backup withholding on checks. See “Deduction plugins” on page 462.
IBulkInvoiceValidationPlugin	Validates a bulk invoice before the bulk invoice submits. This plugin must confirm that a bulk invoice does not violate your business logic. Bulk invoice validation is separate from bulk invoice approval. In general, validation determines whether the data appears to be correct, such as checking whether only certain vendors can submit any bulk invoices. In contrast, submitted invoices go through the approval process with business rules or human approvers to approve or deny an invoice before final processing. See “Bulk invoice integration” on page 445.
IDeductionAdapter	Allows financial deductions to be made from the amount otherwise owed on a payment (for example, back-up withholding or a negotiated discount for a preferred vendor). Called as part of the process of creating a new check. By default, calculates back-up withholding deduction, if any, based on the payee’s tax information in his address book record. See “Deduction plugins” on page 462.
IExchangeRateSetPlugin	To make financial transactions in multiple currencies, ClaimCenter needs a way of describing current currency exchange rates around the world. Do this using the exchange rate set plugin (IExchangeRateSetPlugin) interface, whose main task is to create ExchangeRateSet entities encapsulated in a ExchangeRateSet entity.
OutboundPaymentGatewayPlugin	If the instant payment integration (instant disbursements) feature is enabled in ClaimCenter, this plugin initiates a call to the payment gateway to request an instant payout. See “Payment gateway integration for instant payouts” on page 433.

Plugins for managing ClaimCenter contacts

Plugin Interface	Description
ContactSystemPlugin	Search for contacts and retrieve contacts from an external system. See “Integrating with a contact management system” on page 533.
OfficialIdToTaxIdMappingPlugin	Determines whether ClaimCenter treats an official ID type as a tax ID for contacts. See “Official IDs mapped to tax IDs plugin” on page 295.

Plugins for managing user authentication

Plugin Interface	Description
AuthenticationServicePlugin	Authorizes a user from a remote authentication source, such as a corporate LDAP or other single-source sign-on system.
AuthenticationSource	A marker interface representing an authentication source for user interface login. The implementation of this interface must provide data to the authentication service plugin that you register in ClaimCenter. All classes that implement this interface must be serializable. Any object contained with those objects must be serializable as well. For WS-I web services authentication, see the row in this table for WebservicesAuthenticationPlugin.
AuthenticationSourceCreatorPlugin	Creates an authentication source (an AuthenticationSource) for user interface login. This takes an HTTP protocol request (from an HTTPRequest object). The authorization source must work with your registered implementation of the AuthenticationServicePlugin plugin interface.
DBAuthenticationPlugin	Provides the ability to store the database username and password in a way other than plain text in the config.xml file. For example, retrieve it from an external system, decrypt the password, or read a file from the file system. The resulting

Plugin Interface	Description
	username and password substitutes into the database configuration for each instance of that \${username} or \${password} in the database parameters.
WebservicesAuthenticationPlugin	For WS-I web services only, configures custom authentication logic. This plugin interface is documented with other WS-I information. See “Web services authentication plugin” on page 87.

Plugins for managing document content and metadata

Plugin Interface	Description
IDocumentContentSource	<p>Provides access to a remote document repository for storage and retrieval operations.</p> <p>The example IDocumentContentSource plugin provided in the base configuration must be linked to a commercial document management system before being used in a production environment.</p>
IDocumentMetadataSource	<p>Stores metadata associated with a document, typically in a remote document management system.</p> <p>The example IDocumentMetadataSource plugin provided in the base configuration stores metadata locally and is not intended to be used in a production environment. For maximum data integrity and feature set, implement the plugin and link it to a commercial document management system.</p>

Plugins for managing document production

Plugin Interface	Description
IDocumentProduction	Generates documents from a template. For example, from a Gosu template or a Microsoft Word template. This plugin can create documents synchronously and/or asynchronously. See “Document production” on page 225.
IDocumentTemplateSerializer	<p>This plugin serializes and deserializes document template descriptors. Typically, descriptors persist as XML, as such implementations of this class understand the format of document template descriptors and can read and write them as XML.</p> <p>Use the built-in version of this plugin using the “Plugins registry.” In general, it is best not implement your own version.</p>
IDocumentTemplateSource	Provides access to a repository of document templates that can generate forms and letters, or other merged documents. An implementation may simply store templates in a local repository. A more sophisticated implementation might interface with a remote document management system.

Plugins for message management

Plugin Interface	Description
MessageAfterSend	<p>Optional post-send processing of messages.</p> <p>You can register multiple implementation for this interface to communicate with multiple external systems.</p>
MessageBeforeSend	<p>Optional pre-processing of messages, primarily to transform the payload and return a String that the message transport will use instead. For example, this plugin might convert a flat file of name/value fields to an XML document that is too resource-intensive to create at message creation time.</p> <p>You can register multiple implementation for this interface to communicate with multiple external systems.</p>
MessageReply	Handles asynchronous acknowledgments of a message. After submitting an acknowledgment to optionally handles other post-processing afterward such as property updates. If you can send the message

Plugin Interface	Description
	<p>synchronously, do not implement this plugin. Instead, implement only the transport plugin and acknowledge each message immediately after it sends the message.</p> <p>You can register multiple implementation for this interface to communicate with multiple external systems.</p>
MessageRequest	<p>Optional pre-processing of messages, and optional post-send processing. The MessageRequest interface perform roles of both the related plugins MessageBeforeSend and MessageAfterSend.</p> <p>You can register multiple implementation for this interface to communicate with multiple external systems.</p>
MessageTransport	<p>The main messaging plugin interface within a messaging destination. This plugin sends a message to an external/remote system by using an appropriate transport protocol. This protocol could be a messaging queue, a remote API call, saving special files in the file system, sending emails, and so on.</p> <p>You can register multiple implementation for this interface to communicate with multiple external systems.</p>

Plugins for REST API Framework

Plugin interface	Description
IRestDispatchPlugin	<p>Optional plugin interface to do the following:</p> <ul style="list-style-type: none"> • Preprocess incoming REST API requests • Rewrite outgoing API requests • Control how and what ClaimCenter logs for each API request <p>See the <i>REST API Framework</i> documentation for more information.</p>

Authentication integration

To authenticate ClaimCenter users, the base configuration of ClaimCenter uses the user names and passwords stored in the ClaimCenter database.

IMPORTANT: This configuration provides only a basic level of security using the SHA-1 hash function and a fixed salt.

To provide more secure authentication, authenticate users against an external directory such as a corporate LDAP directory. Alternatively, if ClaimCenter is part of a larger collection of web-based applications, use single sign-on systems to avoid repeated requests for passwords. Using an external directory requires a user authentication plugin.

To authenticate database connections, you might want ClaimCenter to connect to an enterprise database but need flexible database authentication. You might also be concerned about sending passwords as plaintext passwords openly across the network. You can solve these problems with a database authentication plugin. This plugin abstracts database authentication so you can implement it in whichever way your company requires.

Overview of user authentication interfaces

There are several mechanisms to log in to ClaimCenter.

- Log in using the web application user interface.
- Authenticate WS-I web service calls to the current server.

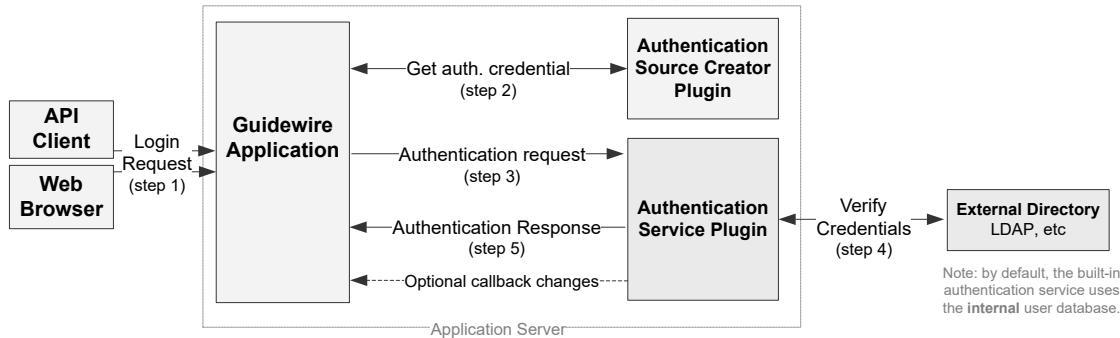
Authentication plugins must handle all types of logins other than WS-I web services.

The authentication of ClaimCenter through the user interface is the most complex, and it starts with an initial request from another web application or other HTTP client. To pass authentication parameters to ClaimCenter, the HTTP request must submit the username, the password, and any additional properties as HTTP parameters. The parameters are name/value pairs submitted within HTTP GET requests as parameters in the URL, or using HTTP POST requests within the HTTP body (not the URL).

Additionally, authentication-related properties can be passed as *attributes*, which are similar to parameters except that they are passed by the servlet container itself, not the requesting HTTP client. For example, suppose Apache Tomcat is the servlet container for ClaimCenter. Apache Tomcat can pass authentication-related properties to ClaimCenter using attributes that were not on the requesting HTTP client URL from the web browser or other HTTP user agent. Refer to the documentation for your servlet container, such as Apache Tomcat, for details of how to pass attributes to a servlet.

The following diagram gives a conceptual view of the steps that occur during login to ClaimCenter.

User Authentication Flow



The chronological flow of user authentication requests is as follows.

1. An initial login request – User authentication requests come from web browsers. If the specified user is not currently logged in, ClaimCenter attempts to log in the user.
2. An authentication source creator plugin extracts the request's credentials – The user authentication information can be initially provided in various ways, such as browser-based form requests or API requests. ClaimCenter externalizes the logic for extracting the login information from the initial request and into a structured credential called an authentication source. This plugin creates the authentication source from information in `HTTPRequests` from browsers and return it to ClaimCenter. ClaimCenter provides a default implementation that decodes username/password information sent in a web-based form. Exposing this as a plugin allows you to use other forms of authentication credentials such as client certificates or a single sign-on (SSO) credentials. In the reference implementation, the PCF files that handle the login page set the username and password as attributes that the authentication source can extract from the request:

```
String userName = (String) request.getAttribute("username");
String password = (String) request.getAttribute("password");
```

3. The server requests authentication using an authentication service – ClaimCenter passes the authentication source to the authentication service. The authentication service is responsible for determining whether or not to permit the user to log in.
4. The authentication service checks the credentials – The built-in authentication service checks the provided username and password against information stored in the ClaimCenter database. However, a custom implementation of the plugin can check against external authentication directories such as a corporate LDAP directory or other single sign-on system.
5. Authentication service responds to the request – The authentication service responds, indicating whether to permit the login attempt. If allowed, ClaimCenter sets up the user session and give the user access to the system. If rejected, ClaimCenter redirects the user to a login page to try again or return authentication errors to the API client. This response can include connecting to the ClaimCenter callback handler, which allows the authentication service to search for and update user information as part of the login process. Using the callback handler allows user profile information and user roles to optionally be stored in an external repository and updated each time a user logs in to ClaimCenter.

User authentication source creator plugin

The authentication source creator plugin (`AuthenticationSourceCreatorPlugin`) creates an authentication source from an HTTP request. The authentication source is represented by an `AuthenticationSource` object and is typically an encapsulation of username and password. However, it also contains the ability to store a cryptographic hash. The details of how to extract authentication from the request varies based on the web server and your other authentication systems with which ClaimCenter must integrate. This plugin is in the `gw.plugin.security` package namespace.

Handling errors in the authentication source plugin

In typical cases, code in an authentication source plugin implementation operates only locally. In contrast, an authentication service plugin implementation typically goes across a network and must test authentication credentials with many more possible error conditions.

Try to design your authentication source plugin implementation so it does not need to throw exceptions. If you do need to throw exceptions from your authentication source plugin implementation, the following guidelines can be useful.

- Typically, the login user interface displays a default general message for failure to authenticate and ignores the text in the actual exception.
- The recommended way to display a custom message with an error is to throw the exception class `DisplayableLoginException`.

```
throw new DisplayableLoginException("The application shows this custom message to the user")
```

Optionally, you can subclass `DisplayableLoginException` to track specific different errors for logging or other reasons.

- If that approach is insufficient for your login exception handling, you can create a custom exception type by subclassing the Java base class `javax.security.auth.login.LoginException`.

See also

- “Create a custom exception type” on page 189

Create a custom exception type

If the login exception types that the base configuration provides do not meet your needs, you can create a custom exception type.

Procedure

1. Create a subclass of `javax.security.auth.login.LoginException` for your authentication source processing exception.
2. To ensure the correct text displays for that class, create a subclass of `gw.api.util.LoginForm`.
3. In your `LoginForm` subclass, override the `getSpecialLoginExceptionMessage(LoginException)` method. ClaimCenter only calls this method for exception types that are not built-in. Your version of the method must return the String to display to the user for that exception type. Note that the only supported method of `LoginForm` for you to override is `getSpecialLoginExceptionMessage`.
4. Modify the `Login.pcf` page, which controls the user interface for that form. That PCF page instantiates `gw.api.util.LoginForm`. Change the PCF page to instantiate your `LoginForm` subclass instead of the default `LoginForm` class.

Authentication data in HTTP attributes

In the base configuration of ClaimCenter, login-related PCF files set the username and password as HTTP request attributes. Unlike URL parameters, HTTP request attributes are hidden values in the request.

The authentication source can extract these attributes from the request in the `HttpServletRequest` object.

```
String userName = (String) request.getAttribute("username");
String password = (String) request.getAttribute("password");
```

The plugin interface provides a single method called `createSourceFromHTTPRequest`. The following example demonstrates how to implement this method.

```
public class BasicAuthenticationSourceCreatorPlugin implements AuthenticationSourceCreatorPlugin {
    public void init(String rootDir, String tempDir) { }

    public AuthenticationSource createSourceFromHTTPRequest(HttpServletRequest request)
```

```

    throws InvalidAuthenticationSourceData {
    AuthenticationSource source;

    // In real code, check for errors and throw InvalidAuthenticationSourceData if errors...
    String userName = (String) request.getAttribute("username");
    String password = (String) request.getAttribute("password");
    source = new UserNamePasswordAuthenticationSource(userName, password);
    return source;
}
}

```

Authentication data in parameters in the URL

If you need to extract parameters from the URL itself, use the `getParameter` method rather than the `getAttribute` method on `HttpServletRequest`.

For example, assume the following URL.

```
https://myserver:8080/cc/ClaimCenter.do?username=aapplegate&password=sheridan&objectID=12354
```

The following code block extracts the `username` and `password` parameter values from the URL.

```

package docexample

uses gw.plugin.security.AuthenticationSource
uses gw.plugin.security.AuthenticationSourceCreatorPlugin
uses gw.plugin.security.UserNamePasswordAuthenticationSource
uses javax.servlet.http.HttpServletRequest

class MyAuthSourceCreator implements AuthenticationSourceCreatorPlugin {
    override function createSourceFromHTTPRequest(request : HttpServletRequest) : AuthenticationSource {
        var userName = request.getParameter( "username" )
        var password = request.getParameter( "password" )
        var source = new UserNamePasswordAuthenticationSource( userName, password )
        return source
    }
}

```

Authentication data in HTTP headers for HTTP basic authentication

The source code for an example `AuthenticationSourceCreatorPlugin` class is provided.

The `BasicAuthenticationSourceCreatorPlugin` class implements HTTP Basic Authentication by retrieving authentication data encoded in an HTTP header.

Java source code for the example class can be found in the `java-examples.zip` file, which is located in the `ClaimCenter` directory.

Extract the files in the archive file. The source files for the example class will be located in the following `ClaimCenter` directory.

```
examples/src/examples/pl/plugins/authenticationsourcecreator
```

The example class decodes a username and password stored in the HTTP request's `Authorization` header and returns a constructed `UserNamePasswordAuthenticationSource` object. The operation is performed in the `createSourceFromHTTPRequest` method.

```

public class BasicAuthenticationSourceCreatorPlugin implements AuthenticationSourceCreatorPlugin {
    public void init(String rootDir, String tempDir) { }

    public AuthenticationSource createSourceFromHTTPRequest(HttpServletRequest request)
        throws InvalidAuthenticationSourceData {
        AuthenticationSource source;
        String authString = request.getHeader("Authorization");
        if (authString != null) {
            byte[] bytes = authString.substring(6).getBytes();
            String fullAuth = new String(Base64.decodeBase64(bytes));
            int colonIndex = fullAuth.indexOf(':');
            if (colonIndex == -1) {
                throw new InvalidAuthenticationSourceData("Invalid authorization header format");
            }
            String userName = fullAuth.substring(0, colonIndex);
        }
    }
}

```

```
String password = fullAuth.substring(colonIndex + 1);
if (userName.length() == 0) {
    throw new InvalidAuthenticationSourceData("Could not find username");
}
if (password.length() == 0) {
    throw new InvalidAuthenticationSourceData("Could not find password");
}
source = new UserNamePasswordAuthenticationSource(userName, password);
return source;
} else {
    throw new InvalidAuthenticationSourceData("Could not find authorization header");
}
}
```

Alternative credentials can be supported by enhancing the method. In such a case, an associated user authentication service that can handle the new source must be implemented in a user `AuthenticationServicePlugin` class.

User authentication service plugin

An `AuthenticationServicePlugin` implementation defines an external service that can authenticate a user. Typically, implementation involves encapsulating authentication credentials in an authentication source and sending the credentials to a separate centralized server on the network such as an LDAP server. This plugin is in the `gw.plugin.security` package namespace.

There are three parts of implementing the `AuthenticationServicePlugin` plugin, each of which is handled by a plugin interface method.

Initialization

All authentication plugins must initialize itself in the plugin's `init` method.

Setting callbacks

A plugin can look up and modify user information as part of the authentication process using the plugin's `setCallback` method. This method provides the plugin with a call back handler of type `CallbackHandler`. Your plugin must save the callback handler reference in a class variable to use it later to make any changes during authentication.

Authentication

Authentication decisions from a username and password are performed in the `authenticate` method. The logic in this method can be almost anything you want, but typically would consult a central authentication database. The basic example included with the product uses the `CallbackHandler` to check for the user within ClaimCenter. The JAAS example calls a JAAS provider to check credentials. Then it looks up the user's public ID in ClaimCenter by username using the `CallbackHandler` to determine which user authenticated.

Every `AuthenticationServicePlugin` must support the default `UserNamePasswordAuthenticationSource`. You can create custom authentication sources with a custom authentication source creator plugin implementation. If you do so, your `AuthenticationServicePlugin` implementation must support your new type of authentication sources.

Almost every authentication service plugin uses the `CallbackHandler` that is provided in the `setCallback` method. The typical use of the callback is to look up the public ID of the user after verifying the credentials. Find the Javadoc for this interface in the class `AuthenticationServicePluginCallbackHandler`. The utility class includes the following methods.

- `findUser` – Looks up a user's public ID based on login user name.
- `modifyUser` – After getting current user data and making changes, perhaps based on contact information stored externally, the method allows the plugin to update the user's information in ClaimCenter.
- `verifyInternalCredential` – Supports testing a username and password against the values stored in the main user database. The method is used by the default authentication service.

Authentication service sample code

The source code for example authentication service plugin classes is provided.

- The `GWAuthenticationServicePlugin` class implements the default authentication service plugin used in the base configuration. The code checks the username and password against information stored in the ClaimCenter database. If the code finds a match in the database, the user is considered to be authenticated.
- The `JAASAuthenticationServicePlugin` class demonstrates how to authenticate against an external repository using JAAS. JAAS is an API that enables Java code to access authentication services without being tied to those services.
- The `LDAPAuthenticationServicePlugin` class demonstrates how to authenticate against an LDAP directory.

Java source code for the example classes can be found in the `java-examples.zip` file, which is located in the `ClaimCenter` directory.

Extract the files in the archive file. The source files for the example classes will be located in the following `ClaimCenter` directory.

```
examples/src/examples/p1/plugins/authenticationservice
```

Handling errors in authentication service plugins

You must be very careful that your code catches and categorizes the different types of errors that can happen during authentication. Preserving this information is important for logging and diagnostic purposes. The base configuration of ClaimCenter provides features to clarify for the user the cause of an authentication failure.

In general, the login user interface displays a default general message for failure to authenticate and ignores the text in the actual exception. You can use `DisplayableLoginException` to show a more specific message to the user from your authentication service plugin code.

The following table lists exception classes that you can throw from your code for various authentication problems.

Exception name	Description
<code>javax.security.auth.login.FailedLoginException</code>	Throw this standard Java exception if the user entered an incorrect password.
<code>gw.plugin.security.InactiveUserException</code>	This user is inactive.
<code>gw.plugin.security.LockedCredentialException</code>	Credential information is inaccessible because it is locked for some reason. For example, a system can be configured to permanently lock out a user after too many failed login attempts. See also the related exception <code>MustWaitToRetryException</code> .
<code>gw.plugin.security.MustWaitToRetryException</code>	The user tried too many times to authenticate and now has to wait for a while before trying again. The user must wait and retry at a later time. This is a temporary condition. See also the related exception <code>LockedCredentialException</code> .
<code>gw.plugin.security.AuthenticationException</code>	Other authentication issues not otherwise specified by other more specific authentication exceptions.
<code>gw.api.util.DisplayableLoginException</code>	This is the only authentication exception for which the login user interface uses the text of the actual exception to present to the user.

You can subclass exception classes in the following ways.

- You can subclass the exception `guidewire.pl.plugin.security.DisplayableLoginException` if necessary for tracking unique types of authentication errors with custom messages.
- If `DisplayableLoginException` does not meet your needs, you can subclass the Java base class `javax.security.auth.login.LoginException`.

See also

- “Create a custom exception type” on page 189

SOAP API user permissions and special-casing users

Guidewire recommends creating a separate ClaimCenter user for SOAP API access. The user or group of users must have the minimum permissions allowable to perform SOAP API calls. Guidewire strongly recommends the user has few or no permissions in the ClaimCenter user interface.

From an authentication service plugin perspective, for those users you could create an exception list in your authentication plugin to implement ClaimCenter internal authentication for only those users. For other users, use LDAP or some other authentication service.

Example authentication service authentication

The following Java code illustrates how to verify a user name and password against the ClaimCenter database and then modify the user's information as part of the login process.

```
public String authenticate(AuthenticationSource source) throws LoginException {
    if (source instanceof UserNamePasswordAuthenticationSource == false) {
        throw new IllegalArgumentException("Authentication source type " + source.getClass().getName() +
            " is not known to this plugin");
    }

    Assert.checkNotNullParam(_handler, "_handler", "Callback handler not set");
    UserNamePasswordAuthenticationSource uNameSource = (UserNamePasswordAuthenticationSource) source;
    Hashtable env = new Hashtable();

    env.put(Context.INITIAL_CONTEXT_FACTORY, "com.sun.jndi.ldap.LdapCtxFactory");
    env.put(Context.PROVIDER_URL, "LDAP://" + _serverName + ":" + _serverPort);
    env.put(Context.SECURITY_AUTHENTICATION, "simple");
    String userName = uNameSource.getUsername();
    if (StringUtils.isNotBlank(_domainName)) {
        userName = _domainName + "\\\" + userName;
    }

    /*
     * The latest revision of LDAPv3 discourages defaulting to unauthenticated
     * when no password is supplied so this should not need checking,
     * however this behavior depends upon the configuration of the LDAP server.
     * See RFC 4511 - inappropriateAuthentication (48) and C.1.12. Section 4.2 (Bind Operation)
     */
    if (StringUtils.isBlank(uNameSource.getPassword())) {
        throw new LoginException("No, or empty, password supplied");
    }

    env.put(Context.SECURITY_PRINCIPAL, userName);
    env.put(Context.SECURITY_CREDENTIALS, uNameSource.getPassword());

    try {
        // Try to login.
        new InitialDirContext(env);
        /* Here would could get the result to the earlier
         * and modify the user in some way if you needed to
         */
    } catch (NamingException e) {
        throw new LoginException(e.getMessage());
    }

    String username = uNameSource.getUsername();
    String userPublicId = _handler.findUser(username);
    if (userPublicId == null) {
        throw new FailedLoginException("Bad user name " + username);
    }

    return userPublicId;
}
```

Deploying user authentication plugins

Like other plugins, there are two steps to deploying custom authentication plugins.

1. Move your code to the proper directory.
2. Register your plugins in the Plugins editor in Studio.

First, move your code to the proper directory. Place your custom `AuthenticationSourceCreator` plugin implementation in the appropriate subdirectory of `ClaimCenter/modules/configuration/plugins/authenticationsourcecreator/basic`, referred to later in this paragraph as `AUTHSOURCEROOT`. For example, if your

class is `custom.authsource.MyAuthSource`, move the location to `AUTHSOURCEROOT/classes/custom/authsource/MyAuthSource.class`. If your code depends on any Java libraries other than the ClaimCenter generated libraries, place the libraries in `AUTHSOURCEROOT/lib/`.

Similarly, place your AuthenticationService plugin in the appropriate subdirectory of `ClaimCenter/modules/configuration/plugins/authenticationservice/basic`, referred to later in this paragraph as `AUTHSERVICEROOT`. For example, if your class is `custom.authservice.MyAuthService`, move the file to the location `AUTHSERVICEROOT/classes/custom/authservice/MyAuthService.class`. If your code depends on any Java libraries other than ClaimCenter generated libraries, place the libraries in `AUTHSERVICEROOT/lib/`.

For ClaimCenter to find and use your custom plugins, you must register them in the plugin editor in Studio. Remember that in the editor, the plugin name and the plugin interface name must match, even though Studio permits you to enter a different value for the plugin name. Otherwise, ClaimCenter does not recognize the plugin. The value for both must be the name of the plugin interface you are trying to use.

The main `config.xml` file contains a `SessionTimeoutSecs` parameter that configures the duration of inactivity to allow before requiring reauthentication. The timeout period controls both the user's HTTP session and the user's session within the ClaimCenter application. Some connection types do not use both an HTTP session and an application session. For example, an API connection to ClaimCenter does not have an HTTP session, only an application session.

Database authentication plugins

Implementing a database authentication plugin provides the following functionality:

- Using a flexible database authentication system to connect the ClaimCenter server to an enterprise database
- Sending passwords across the network without using plain text

Database authentication plugins are different from user authentication plugins. Whereas user authentication plugins authenticate users into ClaimCenter, database authentication plugins help the ClaimCenter server connect to its database server.

A database authentication plugin can retrieve name and password information from the following sources:

- An external system
- A credential provider, by using the implementation of the `CredentialsPlugin` plugin
- A password file on the local file system

The resulting username and password substitutes into the database configuration file anywhere that `#{username}` or `#{password}` are found in the database parameter elements.

The database authentication plugin can also encrypt passwords, and perform other actions.

To implement a database authentication plugin, create a plugin class that implements the class `gw.plugin.dbauth.DBAuthenticationPlugin`.

This class has only one method that you need to implement, `retrieveUsernameAndPassword`, which must return a username and password. Store the username and password combined together as properties on a single instance of the class `UsernamePasswordPair`. The only parameter for `retrieveUsernameAndPassword` is the name of the database for which the application requests authentication information. This name matches the value of the `name` attribute on the `database` or `archive` elements in your `config.xml` file. If you need to pass additional optional properties such as properties that vary by server ID, pass parameters to the plugin in the Guidewire Studio™ configuration of your plugin. Define these parameters in your plugin implementation by using the `@PluginParameter` annotation. The username and password that this method returns are typically not plain text. A plugin like this one typically encodes, encrypts, hashes, or otherwise converts the data into a private format. The only requirement for this format is that the receiver of the data is able to authenticate against the username and password.

The following example demonstrates this method by pulling the information from the `CredentialsPlugin` plugin, if the `CredentialPlugin.Key` plugin parameter has a value, or from text files.

IMPORTANT: This class is provided for demonstration purposes only and is not suitable for a production environment. A production implementation of this plugin must never use an insecure means of storing credentials such as text files.

```
package examples.pl.plugins.dbauthentication;

import aQute.bnd.annotation.component.Activate;
import aQute.bnd.annotation.component.Component;
import aQute.bnd.annotation.component.ConfigurationPolicy;
import gw.pl.util.FileUtil;
import gw.plugin.PluginParameter;
import gw.plugin.Plugins;
import gw.plugin.credentials.CredentialsPlugin;
import gw.plugin.credentials.UsernamePasswordPairBase;
import gw.plugin.dbauth.DBAuthenticationPlugin;
import gw.plugin.dbauth.UsernamePasswordPair;

import java.io.BufferedReader;
import java.io.File;
import java.io.IOException;
import java.util.Map;

/**
 * Simple class that returns a username/password pair with a username and a database password
 * retrieved from the CredentialsPlugin implementation, if available. If not available, retrieves
 * credentials from files specified by the "usernamefile" and "passwordfile" properties.respectively.
 */
@Component(configurationPolicy = ConfigurationPolicy.require)
@PluginParameter(name="passwordfile", type= PluginParameter.Type.File)
@PluginParameter(name="usernamefile", type= PluginParameter.Type.File)
@PluginParameter(name="CredentialPlugin.Key", type=PluginParameter.Type.String,
    helpText = "This is a key to get the username/password from the CredentialPlugin")
public class FileDBAuthPlugin implements DBAuthenticationPlugin {

    private static final String PASSWORD_FILE_PROPERTY = "passwordfile";
    private static final String USERNAME_FILE_PROPERTY = "usernamefile";
    private static final String CREDENTIAL_KEY = "CredentialPlugin.Key";

    private String _passwordfile;
    private String _usernamefile;
    private String _credentialKey;

    @SuppressWarnings("unused")
    @Activate
    void start(Map<?, ?> properties) {
        _passwordfile = (String) properties.get(PASSWORD_FILE_PROPERTY);
        _usernamefile = (String) properties.get(USERNAME_FILE_PROPERTY);
        _credentialKey = (String) properties.get(CREDENTIAL_KEY);
    }

    @Override
    public UsernamePasswordPair retrieveUsernameAndPassword(String dbName) {
        if (_credentialKey != null) {
            CredentialsPlugin plugin = Plugins.isEnabled(CredentialsPlugin.class)
                ? Plugins.get(CredentialsPlugin.class)
                : null;
            if (plugin != null) {
                UsernamePasswordPairBase cred = plugin.retrieveUsernameAndPassword(_credentialKey);
                if (cred != null) {
                    return new UsernamePasswordPair(cred.getUsername(), cred.getPassword());
                }
            }
        }
        try {
            String password = null;
            if (_passwordfile != null) {
                password = readLine(new File(_passwordfile));
            }
            String username = null;
            if (_usernamefile != null) {
                username = readLine(new File(_usernamefile));
            }
            return new UsernamePasswordPair(username, password);
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }

    private static String readLine(File file) throws IOException {
        BufferedReader reader = new BufferedReader(FileUtil.getFileReader(file));
        String line = reader.readLine();
        reader.close();
        return line;
    }
}
```

The base configuration of ClaimCenter includes this example database authentication plugin implementation in the `java-examples.zip` file. The `retrieveUsernameAndPassword` method in this plugin implementation uses the `CredentialsPlugin` plugin, if the `CredentialPlugin.Key` plugin parameter has a value. Otherwise, the method reads a username and password from files specified in the `usernamefile` or `passwordfile` parameters that you define in Studio. ClaimCenter replaces the `${username}` and `${password}` values in the `jdbcURL` parameter with values that your plugin implementation returns. For the source code, refer to the `FileDBAuthPlugin` sample class file in the `examples.pl.plugins.dbauthentication` package.

Configuration for database authentication plugins

For ClaimCenter to find and use your custom plugins, you must register them in the plugin editor in Studio. Remember that in the editor, the plugin name and the plugin interface name must match, even though Studio permits you to enter a different value for the plugin name. Otherwise, ClaimCenter does not recognize the plugin. The value for both must be the name of the plugin interface you are trying to use. Add parameters as appropriate to pass information to your plugin.

To obfuscate the database authentication credentials, see the following:

- *Installation Guide*

WS-I web services authentication

The `WebservicesAuthenticationPlugin` is responsible for authenticating WS-I web services.

In the default implementation, the registered implementation of the `WebservicesAuthenticationPlugin` plugin calls the registered implementation of `AuthenticationServicePlugin` to confirm the name and password. The `AuthenticationServicePlugin` plugin interface handles authentication of web application interactive users.

In typical cases, web service client code sets up authentication and calls desired web services, relying on catching any exceptions if authentication fails. You do not need to call a specific web service as a precondition for login authentication. In effect, authentication happens with each API call. However, if your web service client code wants to explicitly test specific authentication credentials, ClaimCenter publishes the built-in `Login` web service.

Document management

ClaimCenter provides a user interface and integration APIs for creating documents, downloading documents, and producing automated form letters. You can integrate the ClaimCenter document user interface with an external document management system that stores documents and, optionally, the document metadata.

This topic discusses document management in general, as well as details of integrating with a document management system.

The ClaimCenter user interface provides a **Documents** screen when you have a claim open.

The **Documents** screen lists documents such as letters, faxes, emails, and other attachments stored in a document management system (DMS).

Document storage overview

ClaimCenter can store existing documents in a document management system (DMS). For example, you can attach outgoing notification emails, letters, or faxes created by business rules to an insured customer. You can also attach incoming electronic images or scans of paper witness reports, photographs, scans of police reports, or signatures.

The application user interface supports uploading files directly from the browser. The application detects the MIME type of the document based on information from the browser. Note that there are browser differences in this process. For example, RTF files upload with the `application/msword` MIME type on Windows client systems.

Within the ClaimCenter user interface, you can find documents attached to a business object and view the documents. You can also search the set of all stored documents.

Transfer of large documents through the application server to an external document storage system requires significant memory and resources. Even in the best case scenario of memory and CPU resources, the external document storage system or intermediate network can be slow. If so, synchronous actions with large documents can appear unresponsive to a ClaimCenter web user. To address these issues, ClaimCenter provides a way to asynchronously send documents to the document management system without bringing documents into application server memory. For maximum user interface responsiveness with an external document storage system, choose asynchronous document storage. In the default configuration, asynchronous document storage is enabled.

The ContactManager application supports documents attached to vendor contacts only. This application support is part of the integration with ClaimCenter. The support is not used in direct integrations with PolicyCenter or BillingCenter. For use with ClaimCenter, the ContactManager application supports document upload for vendor contact documents but does not support document production.

For document storage, the ContactManager application includes an additional demonstration content storage system implemented by a servlet. This servlet-based content storage system is unsupported for production systems.

Storing document content and metadata

Guidewire recommends integrating with an external document management system (DMS). There are two types of data that a DMS processes for ClaimCenter: document content and document metadata. Each data type can be stored in different locations.

Type of data	Plugin interface and its default behavior
Document content	IDocumentContentSource The base configuration implements an example plugin with classes that store documents on the local file system. The example plugin is for demonstration purposes only and is not intended to be used in a production environment. For example: <ul style="list-style-type: none"> • a fax image • a Microsoft Word file • a photograph • a PDF file A document content source is code that stores and retrieves the document content.
Document metadata	IDocumentMetadataSource The base configuration implements an example plugin in the <code>LocalDocumentMetadataSource</code> class. The example plugin is for demonstration purposes only and is not intended to be used in a production environment. By default, the plugin is disabled in the base configuration. To enable the plugin, clear its Disabled check box in Studio. When the plugin is disabled, document metadata is stored in a local database with a single Document instance for each document. Metadata stored locally provides fast search operations and is available even when the document management system is offline. Alternatively, implement the <code>IDocumentMetadataSource</code> plugin to interact with a remote document management system. A document metadata source is code that manages and searches document metadata. Metadata stored locally by ClaimCenter cannot later be stored by a remote DMS. Similarly, DMS-stored metadata cannot be moved to local ClaimCenter storage. The reason for this restriction is because a text block linked to a document contains a hyperlink. The format of the hyperlink depends on whether the document's metadata is handled by ClaimCenter or a remote DMS. Changing the metadata storage mechanism invalidates the original hyperlink.

Document storage plugin architecture

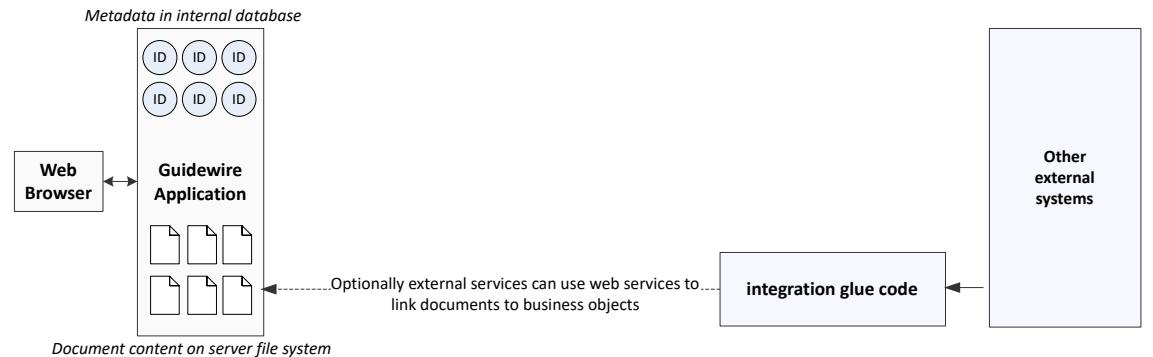
The following diagram summarizes implementation architecture options for document management.

Document Storage Architecture Options

KEY  Document content  Document metadata, such as file name, MIME type, and location of content

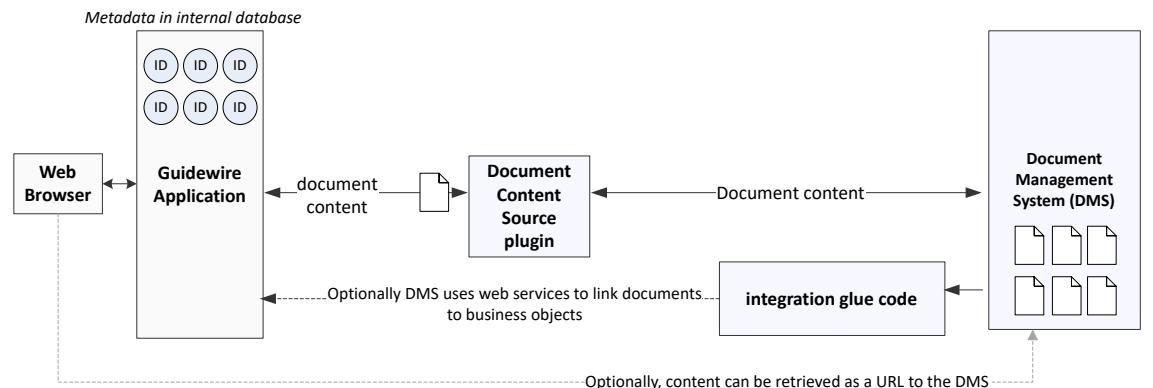
Internal Document Storage and Internal Metadata

Use built-in document content source plugin implementations. Disable the document metadata source plugin



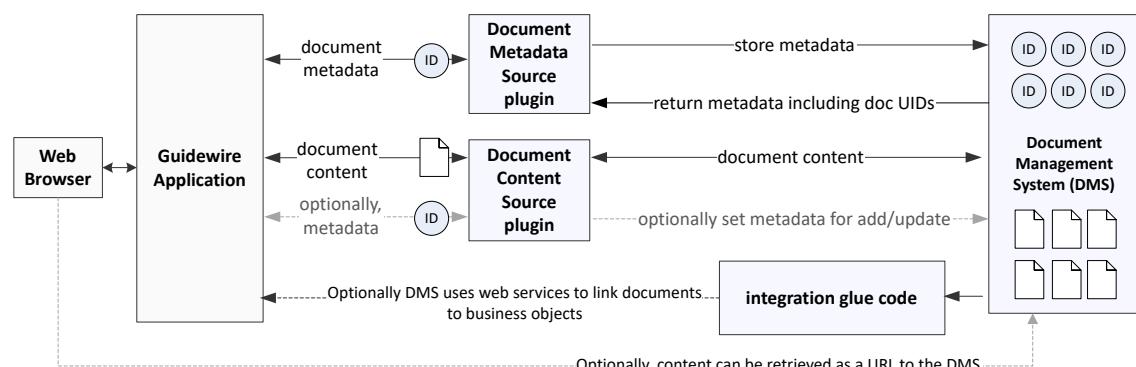
External Document Storage and Internal Metadata (Highest Performance Storage and Searching)

Use customer-written content source plugin implementation. Disable the document metadata source plugin.



External DMS with External Metadata

Use customer-written implementations of both content source plugin and metadata source plugin.



Understanding document IDs

It is important to understand the various types of IDs for a Document entity instance.

- PublicID – The public ID for a single document in the DMS. If the DMS supports versions, the PublicID property does not change for each new version.
 - DocUID – The unique identifier for a single revision of a single document in the DMS. If the DMS supports versions, the DocUID property might change for each new version, depending on how your DMS works.
- The the DocUID property is protected against some types of problems during error conditions. As needed, specific properties including Document.DocUID are copied from one Document entity instance to another. In error conditions, database transaction rollback can in effect cause loss of extension properties because they are not handled specially like the DocUID property. If you have multiple properties that you need to store on the Document entity, serialize them into a single String value and store them in DocUID.
- Your own extension properties that potentially contain IDs – You can extend the base Document entity and add version-related properties or other properties if it makes sense for your DMS. However, there are serious risks in real world usage. In error conditions, database transaction rollback can in effect cause loss of extension properties because they are not handled specially like the DocUID property.
 - Id – Internal ClaimCenter property. Do not get or set the property. In special circumstances, Document.Id is used to identify documents attached to notes and emails.

Document IDs in emails and notes

ClaimCenter supports linking to documents in notes and emails. For emails, the application retrieves any linked documents and includes them as email attachments. For the integration with notes and emails, it is important to note that the use of document IDs varies depending on how you choose to store document metadata. By default, the `IDocumentContentMetadataSource` plugin is disabled in the Plugins registry in Studio. Disabling this plugin causes the application to store document metadata locally in the database with one `Document` entity for each document.

With the `IDocumentContentMetadataSource` plugin disabled, the application uses the natural ID (`Document.Id`) to link to documents inside a note or for email attachments. For notes, this link is persisted in the note content itself. However, for emails the link is temporary until the email is actually sent.

If you register an implementation of the `IDocumentContentMetadataSource` plugin interface so you can store your own document metadata, the behavior is different. The application uses the public ID (`Document.PublicID`) for links to documents within emails and notes.

For outgoing emails, the application only temporarily stores the document public ID at the time of message creation. At a later time in another database transaction (on a server with the messaging server role) at message send time, the application retrieves the document. The server retrieves the document using either `Id` or `PublicID` as specified earlier. This is a rare example of late bound messaging in ClaimCenter. In theory, the contents of the document could have changed between message creation time and message send time. In practice, unless the messaging destination is suspended for long periods of time, the document is unchanged between message creation and message send time.

Implementing a document content source for external DMS

Document storage systems vary in how they transfer documents and how they display documents in the user interface. To support this variety, ClaimCenter supports multiple document retrieval modes called response types.

To implement a new document content source, you must write a class that implements the `IDocumentContentSource` plugin interface. The following sections describe the methods that your class must implement.

This topic includes the following:

- “Check for document existence” on page 201
- “Add documents and metadata” on page 201
- “Retrieve documents” on page 203
- “Update documents and metadata” on page 205
- “Remove documents” on page 205

- “User interface elements when document management system unavailable” on page 205

Check for document existence

A document content source plugin must fulfill requests to check for the existence of a document. To check document existence, ClaimCenter calls the plugin implementation’s `isDocument` method. This method takes a `Document` object as a parameter. Use the `Document.DocUID` property to identify the target document within the repository. If the document with that document UID exists in the external DMS, return `true`. Otherwise return `false`.

The `isDocument` method must not modify the `Document` entity instance in any way.

ClaimCenter calls some methods in a `IDocumentContentSource` plugin—`isDocument` and `getDocumentContentsInfo`—twice for each document. Your plugin implementation must support this design.

The `isDocument` function is called before the rendering of the `View` button on the Documents page. If `isDocument` returns `false`, then the `View` button does not call the document content source plugin to open the document. Therefore, for all existing documents for which the end user wants the `View` button to display the document, the `isDocument` implementation necessarily must return `true`. In addition, the DMS bit for those documents must have a value of `true`.

Note: ClaimCenter does not require the document name to be unique in the DMS. If you cannot make the document unique, there is an API that you can use or modify to adjust the name. In Studio, see the `adjustDocumentNameIfDuplicate` method in the `gw.document.DocumentEnhancement` enhancement.

Distinction between `isDocument` function return value and DMS bit value

The `isDocument` function return value differs in meaning from the meaning of the value of the DMS bit on a `Document` object. The `isDocument` function return value indicates whether the external DMS has a `Document` object with a particular `DocUID` property value. If the `isDocument` function returns `true`, a matching `Document` object exists. In this case, what the matching object contains depends in part on whether the `IDocumentMetadataSource` (IDMS) plugin is defined and in part on the value of the DMS bit. The reason for two different flags—the `isDocument` function return value and the DMS bit value—is the interaction between the IDMS plugin and contentless documents.

In the case that the `isDocument` function returns `true`, whether the matching `Document` object contains metadata in the external DMS will depend on whether the IDMS plugin is defined. If the IDMS plugin is not defined, the matching `Document` object in the external DMS can have only content. Corresponding metadata for the matching object will exist only in the ClaimCenter database, not the external DMS. If the IDMS plugin is defined, the matching object can have either or both corresponding content and corresponding metadata in the external DMS.

Again in the case that the `isDocument` function returns `true`, the DMS bit indicates whether what the external DMS stores for the corresponding `Document` object includes actual content. If the DMS bit is `true`, the external DMS stores actual content. If `false`, the external DMS stores no actual content.

For example, suppose that the return value for the `isDocument` function is `true`. Suppose in addition that the IDMS plugin is defined. Further suppose that the value of the matching `Document` object’s DMS bit is `false`. In this case, the external DMS stores metadata for the `Document` object identified as a parameter for the `isDocument` function but no actual content.

Note: It is possible for the `isDocument` function to return `false`, an IDMS plugin to exist, and for the DMS bit to be `true`. In this case, a `Document` object that has either or both content and metadata can be in transit to without having reached the external DMS. This scenario can happen if the implementations of the `IDocumentMetadataSource` (IDMS) and `IDocumentContentSource` (IDCS) plugins are asynchronous.

Add documents and metadata

A new document content source plugin must fulfill a request to add a document by implementing the `addDocument` method. The method adds a new document object to the repository.

If you enable and implement the `IDocumentMetadataSource` plugin, ClaimCenter also calls the `IDocumentContentSource` plugin’s `addDocument` method to update an existing document. Write your `addDocument` method to expect to be called with an existing document.

The method takes the following arguments.

- The document metadata in a Document object.
- The document contents as an input stream (`InputStream`) object. The input stream could be empty.

If the input stream reference is `null` in your `addDocument` method, this has a special meaning. You have already sent the document to the DMS, but you have an opportunity to perform actions after rollback after some types of errors.

In a real document, the `Document.InputStream` property can contain the value `null`, which represents what is called a contentless document. An example of a contentless document is a document located in a filing cabinet, but not scanned electronically. In ClaimCenter, the user interface represents this by the document type Indicate Existence. However, ClaimCenter never calls the `IDocumentContentSource` method `addDocument` for contentless documents. If the input stream argument to `addDocument` is `null`, it always represents an opportunity to run code related to error recovery.

The `addDocument` method also may copy some of the metadata contained in the `Document` entity into the external system. Independent of whether the document stores any metadata in the external system, the `addDocument` method can update certain metadata properties within the `Document` object.

- The method must set an implementation-specific document universal ID in the `Document.DocUID` property.
- For ClaimCenter, the method must set the modified date in the `Document.DateModified` property. For other applications including ContactManager, setting the modified date is optional.

The code that calls your `addDocument` method attempts to persist your changes during the final bundle commit. The bundle contains your changes to the `Document` entity instance and potentially other changes to other entity instances made before or after the application calls `addDocument`.

If errors occur that prevent the modified `Document` entity instance from persisting to the database, by default your changes to the `Document.DocUID` property are preserved during rollback. You can optionally configure additional behavior after rollback.

Return value from the addDocument method

The return value from the `addDocument` method is very important.

- If you do not enable and implement the `IDocumentMetadataSource` plugin, you must return `false`.
- If you enable and implement the `IDocumentMetadataSource` plugin implementation, the following conditions must be met.
 - If you stored or updated metadata from the `Document` object in addition to document content, return `true`.
 - Otherwise, return `false`. ClaimCenter calls the registered `IDocumentMetadataSource` plugin implementation to store or update the metadata.

What happens for errors and validation issues after you send files to the DMS

Normally, ClaimCenter persists changes to the `Document` entity that you made in `addDocument`. For example, your `addDocument` implementation must set `Document.DocUID` to a value that can be used to retrieve it from the DMS. However, even if the `addDocument` method completes successfully without throwing exceptions, errors could still occur in other code before the `Document` entity instance changes persist to the database. Even what appears to the user as a validation warning (not error) would prevent the bundle from committing. Any recent changes to the bundle including the `Document` entity are rolled back, which means to be restored to their original values in the database.

During rollback, your changes to the `Document.DocUID` property and only that property are preserved. Before rollback occurs, the application saves the `Document.DocUID` property value. Note that the `Document.DocUID` property value can optionally contain a serialized set of multiple custom properties in one `String` value. Then, the application restores the entity instance to its previous state in the database (this is the rollback). Finally, the application sets the `Document.DocUID` property value to the previously saved value, which might include changes that your code has made. However, no other properties from the previous database transaction are restored. The `Document.DocUID` property is preserved to reduce the chance of orphaned documents, which are documents sent to the DMS that have no persisted links from ClaimCenter objects.

You might want multiple custom fields on the `Document` entity instance to survive rollback after your code completes its `addDocument` method. If so, it is best to remove the custom fields from the `Document` data model and serialize that information collectively into the `DocUID` property. This technique provides some level of protection during error conditions that cause a rollback because the `DocUID` property is specially preserved.

It is important to understand that after rollback, the `Document` entity instance is in ClaimCenter server memory but not yet persisted. If the user fixes whatever errors caused the rollback (such as validation issues) and tries again, the `Document` entity instance is persisted along with the rest of the bundle. However, if the bundle is never committed due to repeated failure to commit the bundle, the document content is orphaned. Your code has sent the document content to the DMS, but no metadata or ClaimCenter object links to that document content.

There is an additional optional configuration of `Document` rollback that might be useful in some cases. There is a `config.xml` property called `RecallDocumentContentSourceAfterRollback`. By default it is set to `false`. If you set it to `true` and errors occur after your `addDocument` method is called but before the bundle is committed, the application calls `addDocument` again after rollback. However, for this circumstance, the input stream argument to `addDocument` is `null`. The `null` value for the input stream indicates that you do not need to send document content to the DMS because it succeeded already. However, your `addDocument` method has an opportunity to do additional processing of the `Document` entity or communicate with the DMS after rollback, if it is useful to do so.

Error handling during the `addDocument` method

If your plugin code encounters errors before returning from this method, throw an exception from the `addDocument` method. Remember that if you support asynchronous document storage, the original action that triggered the document content storage may have already completed.

Your document content storage plugin must handle errors in an appropriate fashion. For example, send administrative emails or create a new activity to investigate the problem. You could optionally persist the error information in a separate database transaction and create an administration system to review the issues.

Retrieve documents

An implementation of the `IDocumentContentSource` plugin must fulfill requests to retrieve a document. The following two plugin methods perform this task.

```
getDocumentContentsInfo(document : Document, includeContents : boolean) : DocumentContentsInfo  
getDocumentContentsInfoForExternalUse(document : Document) : DocumentContentsInfo
```

In typical cases when a document must be retrieved, the `getDocumentContentsInfo` method is called. In cases where the retrieved document will be published externally, such as when it will be attached to an outgoing email, the `getDocumentContentsInfoForExternalUse` method is called.

Arguments

Both retrieval methods accept a `Document` object argument. The `DocUID` property specifies the desired document.

The plugin methods can access the `Document` argument's properties, but the properties must not be modified. To add modifiable properties to the `Document` class, the recommended method is to serialize the additional data in the `DocUID` property. This technique is preferred over the usual practice of adding properties by extending the `Document` class.

The `includeContents` argument specifies whether to include the document contents in the returned object.

The `getDocumentContentsInfo` method can return a reference to the retrieved document in the following formats.

- As a URL address to a network store that can display the document contents
- As an input stream that contains the document contents as a stream of bytes

The `getDocumentContentsInfoForExternalUse` method always returns the retrieved document as an input stream that contains the document contents.

Return value

Both retrieval methods return a `DocumentContentsInfo` object containing the response data. The following properties of the object can be set.

- `ResponseType` – Specifies the format of the retrieved document contents. Note that the actual document contents are returned only if the method's `includeContents` argument is true. However, the `ResponseType` property is set regardless of whether the document contents are returned. If the contents are not returned, the property specifies the format the contents would have been returned in.

Supported response type values are `DOCUMENT_CONTENTS`, `URL`, and `URL_DIRECT`. Each value is referenced through `gw.document.DocumentContentsInfo.ContentResponseType`, as shown in the example below.

```
returnedDocContentsInfo.ResponseType = gw.document.DocumentContentsInfo.ContentResponseType.URL
```

The supported response types are described below.

- `DOCUMENT_CONTENTS` – If document contents are returned then the `DocumentContentsInfo.InputStream` property references an input stream that contains the document contents as a stream of bytes.
- `URL` – If document contents are returned, the `DocumentContentsInfo.InputStream` property contains the relevant URL address. The plugin caller will read the address, open a connection to it, and retrieve the document contents. The contents are then returned to ClaimCenter and the client browser just as if the `DOCUMENT_CONTENTS` response type had been specified.
- `InputStream` – Set only if the method's `includeContents` argument is true. Otherwise, undefined. If set, the `InputStream` contents are determined by the value of the `ResponseType` property.
- `ContentDispositionType` – Set only if the method's `includeContents` argument is true. Otherwise, undefined. Specifies the type of disposition for the document content downloads. The property can contain one of the following values from the `ContentDispositionType` enumerator.
 - `DEFAULT` – Use the document disposition specified by the `DocumentContentDispositionMode` configuration parameter.
 - `INLINE` – Request the client browser to load and, optionally, render the document.
 - `ATTACHMENT` – Request the client browser to download the document. Optionally, launch an external program to render the document.
- `ResponseMimeType` – Never set this property. ClaimCenter will set the property itself based on the properties of the retrieved document.

Retrieve a document's content disposition

The `DocumentsUtilBase` class provides a convenience method that retrieves the content disposition of a document.

```
getContentDispositionForDocument(document : Document) : ContentDisposition
```

The method accepts an argument of the relevant `Document`. The `DocUID` property identifies the specific document.

The method calls the `IDocumentContentSource` plugin method `getDocumentContentsInfo` to retrieve the document's information, including its content disposition.

If either a problem occurs in the retrieval operation or the retrieved disposition value is `DEFAULT`, the method returns the `ContentDisposition` value specified in the `DocumentContentDispositionMode` configuration parameter. Otherwise, the method returns the document's `ContentDisposition` value as specified in the `DocumentContentsInfo` class `ContentDispositionType` property.

See also

- “Render a document” on page 206

Update documents and metadata

A new document content source plugin must fulfill requests from ClaimCenter to update documents. To update documents, implement the `updateDocument` method. This method takes two arguments.

- A replacement document, as a stream of bytes in an `InputStream` object.
- A `Document` object, which contains document metadata. The `Document.DocUID` property identifies the document in the DMS.

At a minimum, the DMS system must update any core properties in the DMS that represent the change itself, such as the date modified, the update time, and the update user. If the document `UID` property implicitly changes because of the update, set the `DocUID` property on the `Document` argument. ClaimCenter persists changes to the `Document` object to the database.

Optionally, your document content source plugin can update other metadata from `Document` properties. The DMS must update the same set of properties as in your document content source plugin `addDocument` method. If your DMS has a concept of versions, you can extend the base `Document` entity to contain a property for the version. If you extend the `Document` entity with version information, `updateDocument` method sets this information as is appropriate.

During document update, you can optionally set `Document.DocUID` and any extension properties that represent versions. However, do not change the `Document.PublicID` property.

The return value from the `updateDocument` method is very important.

- If you do not enable and implement the `IDocumentMetadataSource` plugin, you must return `true`.
- If you enable and implement the `IDocumentMetadataSource` plugin implementation, the following conditions must be met.
 - If you stored metadata from the `Document` object in addition to document content, return `true`.
 - Otherwise, return `false`. ClaimCenter calls the registered `IDocumentMetadataSource` plugin implementation to store the metadata.

Remove documents

A new document content source plugin must fulfill a request to remove a document. To remove a document, ClaimCenter calls the plugin implementation's `removeDocument` method. This method takes a `Document` object as a parameter. Use the `Document.DocUID` property to identify the target document within the repository.

ClaimCenter calls this method to notify the document storage system that the document corresponding to the `Document` object will soon be removed from metadata storage. Your `IDocumentContentSource` plugin implementation can decide how to handle this notification. Choose carefully whether to delete the content, retire it, archive it, or take another action that is appropriate for your company. Other `Document` entities may still refer to the same content with the same `Document.DocUID` value, so deletion may be inappropriate.

Be careful writing your document removal code. If the `removeDocument` implementation does not handle metadata storage, then a server problem might cause removal of the metadata to fail even after successfully removing document contents.

If the `removeDocument` method removed document metadata from metadata storage and no further action is required by the application, return `true`. Otherwise, return `false`. If you do not enable and implement the `IDocumentMetadataSource` plugin, you must return `true`.

User interface elements when document management system unavailable

The user interface is generally responsive to the presence of a document management system. However, certain user interface controls show inappropriate status or are disabled when a configured document management system is not available. This may include usually actionable controls being unavailable, actionable controls that inexplicably do not take action, or conflicts between actionable and non-actionable user interface controls.

You can check the availability of the document management system through the `gw.document.DocumentsActionsUIHelper.DocumentContentServerAvailable` property. Check this property as appropriate to improve the user experience when the document management system is unavailable.

Render a document

The `DocumentsUtilBase` class provides various methods to assist the user interface in rendering the contents of a retrieved document.

```
renderDocumentContentsWithDownloadDisposition(filename : String,
                                              documentContentsInfo : DocumentContentsInfo)

renderDocumentContentsDirectly(filename : String,
                               documentContentsInfo : DocumentContentsInfo,
                               contentDisposition : ContentDisposition)
```

Both methods render a referenced document to a file, which the client browser can process.

The `filename` argument specifies the file in which to render the document. The `documentContentsInfo` argument contains information about the retrieved document. Each method requires the document's `ContentDisposition` value. The disposition is passed to the `renderDocumentContentsDirectly` method in the `contentDisposition` argument. The `renderDocumentContentsWithDownloadDisposition` method retrieves the disposition from the `Download` attribute of the current user interface widget.

The `DocumentsUtilBase` class also provides methods that are intended to be used by a user interface Gosu action. Each method marks a specified `FileInput` file widget so that the server executes a special render command during processing of the response. The command instructs the client browser to download the contents of the relevant document and immediately display the downloaded contents in the widget.

```
markFileInputForClientInitiatedDownload(fileWidgetId : String)
markFileInputForClientInitiatedDownload(fileWidgetId : String,
                                         contentDisposition : ContentDisposition)
markFileInputsForClientInitiatedDownload(Object[] fileDownloadSpecs)
```

Each of the two signatures for the `markFileInputForClientInitiatedDownload` method accepts a `fileWidgetId` argument that specifies the ID of a `FileInput` widget. Each signature requires a `ContentDisposition` value. One signature accepts a `contentDisposition` argument, while the other signature always uses the disposition specified in the `DocumentContentDispositionMode` configuration parameter.

As its slightly different name implies, the `markFileInputsForClientInitiatedDownload` method can mark multiple `FileInput` widgets. The method accepts an array of objects where each object is one of the following elements:

- A `FileInput` ID in the form of a `String`. The disposition specified in the `DocumentContentDispositionMode` configuration parameter is used to process the document.
- A two-element array containing a `FileInput` ID and the `ContentDisposition` value to process the document.

The following example code demonstrates the method's Gosu syntax using various arguments.

```
markFileInputsForClientInitiatedDownload({
    'widgetID_001',                                // Use DocumentContentDispositionMode config param
    {'widgetID_002', gw.document.ContentDisposition.DEFAULT}, // Also uses the config parameter
    {'widgetID_003', gw.document.ContentDisposition.INLINE},
    {'widgetID_004', gw.document.ContentDisposition.ATTACHMENT}
})
```

Implementing an `IDocumentMetadataSource` plugin

Document metadata is typically handled by the `IDocumentMetadataSource` plugin. In the base configuration, the plugin is disabled, which causes metadata to be stored locally by ClaimCenter. To store metadata remotely using a data management system, implement the `IDocumentMetadataSource` plugin to interact with the DMS.

The `IDocumentMetadataSource` plugin tracks a document by using the `PublicID` property, not the `DocUID` property. The `DocUID` property primarily relates to the document content, not its metadata.

With the `IDocumentMetadataSource` plugin enabled, asynchronous document storage is not supported.

This topic includes the following:

- “Basic methods for the `IDocumentMetadataSource` plugin” on page 207
- “Linking methods for the `IDocumentMetadataSource` plugin” on page 209

Basic methods for the `IDocumentMetadataSource` plugin

This topic covers the following methods:

- “`saveDocument`” on page 207
- “`retrieveDocument`” on page 207
- “`searchDocuments`” on page 208
- “`removeDocument`” on page 208
- “`isOutboundAvailable`” on page 209
- “`isInboundAvailable`” on page 209

saveDocument

Implement the `saveDocument` method to persist document metadata from a `Document` object to your document repository. The only argument is a `Document` entity. If document content source plugin methods `addDocument` or `updateDocument` return `false` to indicate they did not handle metadata, ClaimCenter calls the document metadata source plugin `saveDocument` method.

If the document or any child object does not already have a non-null `PublicID` property, the method must select and set a unique `PublicID` value.

The method has no return value.

When using `IDocumentMetadataSource` plugin, messaging events never fire for document actions. If your application logic requires to be notified about document change events, add any relevant logic to the relevant plugin methods. The typical location to add this kind of logic is the `IDocumentContentSource` plugin implementation, not the `IDocumentMetadataSource` plugin implementation.

retrieveDocument

The `retrieveDocument` method retrieves a document’s metadata from a document management system.

```
retrieveDocument(uniqueId : String) : Document
```

The `uniqueId` argument specifies the document’s unique public ID. If an error occurs when communicating with the document management system, the method throws a `java.rmi.RemoteException`.

The method returns a `Document` object containing the retrieved metadata. In addition to containing the retrieved metadata, the internal properties of the `Document` object must be initialized by calling the following methods:

- `DocumentsUtilBase.initOriginalValues` – The method accepts the `Document` object as an argument.
- The `Document` object’s `setRetrievedFromIDMS` method – The method takes no arguments. It performs the following operations:
 - Marks the `Document` as having been initialized from the document management system.
 - Initializes the `Document` entity’s internal flag indicating that the `Document` does not currently need to be persisted.

The following example code demonstrates the required initialization for a retrieved `Document` object.

```
var retrievedDoc : Document
```

```
// ... Create Document object and initialize with the data retrieved from the DMS  
// Before returning, initialize the Document object's internal properties  
DocumentsUtilBase.initOriginalValues(retrievedDoc)  
retrievedDoc.setRetrievedFromIDMS()
```

searchDocuments

Implement the `searchDocuments` method to return the set of documents in the repository that match the given set of criteria. It is important to understand that it is the responsibility of your plugin implementation to properly filter and order the results based on the arguments to the `searchDocuments` method.

One method argument is a `DocumentSearchCriteria` entity instance, which defines the search criteria to send to the DMS. Refer to the “*Data Dictionary*” for details about the individual search criteria properties in an `DocumentSearchCriteria` entity instance.

Another argument is a `RemotableSearchResultSpec` Java object. This object contains sorting and paging information for the PCF list view infrastructure for the results.

- `GetNumResultsOnly` – A boolean value that specifies whether to return only the number of results
- `SortColumns` – An array of `RemotableSortColumn` objects that define the sort order
- `StartRow` – The start row, as an `int` value
- `MaxResults` – Maximum results to return at one time, as an `int` value
- `IncludeTotal` – A boolean value that specifies whether to return the number of search results in the `TotalResults` property.

When a user searches for documents, ClaimCenter calls the `searchDocuments` plugin method twice. The first invocation retrieves the number of results. The second invocation gets the results. The two invocations are differentiated by the value in the `GetNumResultsOnly` argument.

Before returning from this method, you must call the new `setRetrievedFromIDMS` method on each `Document` entity instance. The `setRetrievedFromIDMS` method performs the following actions.

- Sets the `Document` property `PersistenceRequired` to `false`.
- Marks the document as coming from the `IDocumentMetadataSource` plugin.

The `searchDocuments` method must return search results in a `DocumentSearchResult` entity instance. Create a new entity instance. To add a search result, call the `addToSummaries` method on the `DocumentSearchResult` entity instance and pass a `Document` entity instance. Depending on how your DMS implementation implements search, you might directly create the `Document` entity instance or you might call your own `IDocumentMetadataSource` plugin method `retrieveDocument`. You can optionally set the `Summaries` property on `DocumentSearchResult` to an array of `Document` objects.

If the `IncludeTotal` or `GetNumResultsOnly` property is `true` on the `RemotableSearchResultSpec` object, the method sets the number of results in the `TotalResults` property.

removeDocument

Implement the `removeDocument` method to remove document metadata for a `Document` object from the document repository. You do not need actually to delete a document in the DMS. Instead, you can mark the document metadata so that it does not appear if the Guidewire application searches for this document metadata.

The method returns nothing.

When using `IDocumentMetadataSource` plugin, messaging events never fire for document actions. If your application logic requires notification about document change events, add any relevant logic to the relevant plugin methods. The typical location to add this kind of logic is the `IDocumentContentSource` plugin implementation, not the `IDocumentMetadataSource` plugin implementation.

isOutboundAvailable

Implement the `isOutboundAvailable` method. From Gosu this is the `OutboundAvailable` property. Return `true` to indicate that the external DMS is available for storing metadata. If this returns `true`, the following methods can be called: `isDocument` and `getDocumentContentsInfo`.

isInboundAvailable

Implement the `isInboundAvailable` method. From Gosu this is the `InboundAvailable` property. Return `true` to indicate that the external DMS is available for retrieving metadata. If the method returns `true`, the following methods can be called.

- `addDocument`
- `updateDocument`
- `removeDocument`

Linking methods for the `IDocumentMetadataSource` plugin

The `IDocumentMetadataSource` plugin includes methods that enable the creation of many-to-many links from a `Document` object to various Guidewire entity types.

Many-to-many document links are supported by ClaimCenter and ContactManager only.

PolicyCenter and BillingCenter support one-to-one links where a `Document` object can link to a single instance of a particular entity type. The methods described in this section do not apply to PolicyCenter or BillingCenter.

In ClaimCenter, many-to-many document links are not supported to all Guidewire entity types. A ClaimCenter `Document` object can link to only a single instance of the following entity types.

- `Claim`
- `ClaimContact`
- `Exposure`
- `Matter`

A ClaimCenter `Document` object can have many-to-many links to the following entity types.

- `Activity`
- `CheckSet`
- `ReserveSet`
- `ServiceRequest`

In ContactManager, a `Document` object can have many-to-many links only to instances of the `ABContact` entity type.

This topic covers the following methods:

- “`linkDocumentToEntity`” on page 209
- “`getDocumentsLinkedToEntity`” on page 210
- “`isDocumentLinkedToEntity`” on page 210
- “`unlinkDocumentFromEntity`” on page 210
- “`getLinkedEntities`” on page 210

linkDocumentToEntity

Implement the `linkDocumentToEntity` method to associate a document with a Guidewire entity. The arguments are an entity instance and a `Document` entity instance.

The method has no return value.

Note that document actions do not trigger a messaging event. If your application logic requires notification about document change events, add the necessary logic to the relevant plugin methods. However, in most situations, this kind of logic is added to the `IDocumentContentSource` plugin implementation, not the `IDocumentMetadataSource` plugin.

getDocumentsLinkedToEntity

Implement the `getDocumentsLinkedToEntity` method to return all documents associated with a Guidewire entity. The only argument is an entity instance. Return a `DocumentSearchResult` entity instance, which encapsulates a list of documents. The `DocumentSearchResult` entity instance is the same type of object as returned by the `searchDocuments` method.

This method must call the `setRetrievedFromIDMS` method on any new `Document` entity instance. The `setRetrievedFromIDMS` method sets `document.PersistenceRequired` to `false` and marks the document as coming from the `IDocumentMetadataSource` plugin.

isDocumentLinkedToEntity

Implement the `isDocumentLinkedToEntity` method to check if a document is associated with a Guidewire entity instance. Return `true` if the document is associated with a Guidewire entity instance. Otherwise, return `false`.

unlinkDocumentFromEntity

Implement the `unlinkDocumentFromEntity` method to remove the association between a document and a Guidewire entity.

The method has no return value.

getLinkedEntities

The `getLinkedEntities` method is supported by ContactManager only.

Implement the `getLinkedEntities` method to get a list of public IDs for entity instances associated with a specific document.

Document storage plugins in the base configuration

In the base configuration of ClaimCenter, the class `gw.plugin.document.impl.AsyncDocumentContentSource` implements the `IDocumentContentSource` plugin. This plugin implementation stores documents on the ClaimCenter server's file system. By default, ClaimCenter stores the document metadata, such as document file names as the document names, in the database. You can override this behavior by writing a class that implements the `IDocumentMetadataSource` plugin and then registering that class in `IDocumentMetadataSource.gwp`.

In the base configuration, Guidewire ContactManager supports documents attached to vendor contacts, maintaining the connection of document to vendor and maintaining the vendor information. This support is part of the integration with ClaimCenter and is not used in base configuration integrations with PolicyCenter or BillingCenter.

For document storage, the base configuration of ContactManager provides a content storage system implemented by a servlet. This servlet-based content storage system is not supported for production systems.

This topic includes the following:

- “Documents storage directory and file name patterns” on page 210
- “Remember to store public IDs in the external system” on page 211

Documents storage directory and file name patterns

In the base configuration, the document content and storage plugins registered in the Plugins editor use the `documents.path` plugin parameter to specify the storage location. Plugin parameters are a set of name/value assignments in the Plugins registry file (the `.gwp` file) for that plugin interface.

For example, open Guidewire Studio and, in the **Project** window, navigate to **configuration > config > Plugins > registry**. Then open **IDocumentContentSource.gwp** in the Plugins Registry editor. In the base configuration, the plugin implementation class **gw.plugin.document.impl.AsyncDocumentContentSource** is registered. In the **Parameters** section, the following parameters are defined:

```
documents.path  
  Value: files\documents  
  
TrySyncedAddFirst  
  Value: true  
  
SyncedContentSource  
  Value: gw.plugin.document.impl.LocalDocumentContentSource
```

For production systems, you must set the **documents.path** plugin parameter to an absolute file path. For clusters of ClaimCenter servers, you must use a network file share so that other servers can see document contents that are written by other servers. Set the absolute path to a network file share that you mount at the same location in the local file system for all members of the cluster.

If you are running a development or testing server with no need for persistence across server restart, you can use a relative path for **documents.path**. If that plugin parameter value is a relative path, then the location is determined by using the value of the temporary directory parameter (**javax.servlet.context.tempdir**). The temporary directory property is the root directory of the servlet container's temp directory structure. This is a directory that servlets can write to as scratch space but with no guarantee that files persist from one session to another.

IMPORTANT: Never rely on temporary directory data in a production ClaimCenter system.

Documents are partitioned into subdirectories by claim and by relationships within the claim. The following table explains this directory structure.

Entity	Directory naming	Example
Claim	Claim + claimNumber	/documents/Claim02-02154/
Exposure	claimDirectory/Exposure + exposurePublicID	/documents/Claim02-02154/ExposureABC:01/
Claimant	claimDirectory/Claimant + claimantPublicID	/documents/Claim02-02154/ClaimantABC:99/

Note: For exposure and claimant, the public ID is used, whereas for a claim the claim number is used.

ClaimCenter stores documents by the name you choose in the user interface as you save and add the file. If you add a file from the user interface that would result in a duplicate file name, ClaimCenter warns you. The new file does not quietly overwrite the original file. If you create a document by using business rules, the system forces uniqueness of the document name. The server appends an incrementing number in parentheses. For example, if the file name is **myfilename**, the duplicates are called **myfilename(2)**, **myfilename(3)**, and so on.

The document metadata source plugin in the base configuration provides a unique document ID back to ClaimCenter. That document ID identifies the file name and relative path in the repository to the document.

Following is an example repository's relative path to an exposure document:

/documents/claim02-02154/exposureABC:01/myFile.doc

Remember to store public IDs in the external system

In addition to the unique document IDs, remember to store the **PublicID** property for Guidewire entities, such as **Document**, in external document management systems.

ClaimCenter uses public IDs to refer to objects if you later search for the entities or re-populate entities during search or retrieval. If the public ID properties are missing on document entities during search or retrieval, the ClaimCenter user interface may have undefined behavior.

Asynchronous document storage

Some document production systems generate documents slowly. When many users try to generate documents at the same time, multiple CPU threads compete and that makes the process even slower. One alternative is to create documents asynchronously so that user interaction with the application does not block waiting for document production.

Similarly, transfer of large documents through the application server to an external document storage system requires significant memory and resources. Even in the best case scenario of memory and CPU resources, the external document storage system (or intermediate network) may be slow. If the system is slow, synchronous actions with large documents may appear unresponsive to a ClaimCenter user.

To address these issues, ClaimCenter provides a way to asynchronously send documents to the document management system without bringing documents into application server memory. On servers with the **messaging** server role, a separate thread sends documents to your real document management system using the messaging system.

If a document is sent asynchronously, a document is temporarily in the *pending* state. For a pending document, there may be a **Document** entity instance but the file is not yet sent to the DMS. Be aware that some documents may not appear in the user interface until after the document is completely sent to the DMS. In some application contexts, the document might appear but is neither viewable nor editable.

Whether you use asynchronous or synchronous document storage to connect to an external document management system, your main task is the same. You must implement your own **IDocumentContentSource** plugin implementation. There are configuration differences depending on whether you want to support asynchronous document storage.

ClaimCenter includes the following code to support this feature.

- The built-in asynchronous document content source – Instead of directly registering your own **IDocumentContentSource** plugin implementation in the plugin registry, register the built-in document content source plugin implementation `gw.plugin.document.impl.AsyncDocumentContentSource`. By default, the asynchronous document storage plugin is enabled. When this document content source gets a request to send the document to the document management system, it immediately saves the file to a temporary directory on the local disk. In a clustered environment, you can map the local path to this temporary directory so that it references a server directory shared by all elements of the cluster. For example, map this to an SMB server.
- The built-in document storage messaging transport – A separate system uses ClaimCenter messaging to send documents to the external system. In a clustered environment, any server can create a new message. However, in the base configuration, only servers in the cluster with the **messaging** server role instantiate the messaging plugins to send messages across the network. It is the asynchronous document transport that calls the document storage plugin implementation that you implement. As a result, in a clustered environment, your document content source implementation runs on servers with the **messaging** server role.

For maximum document data integrity and document management features, use a complete commercial document management system.

In ClaimCenter and ContactManager, it is unsupported to use the asynchronous document storage plugin in conjunction with enabling the **IDocumentMetadataSource** plugin. To enable the **IDocumentMetadataSource** plugin, you must disable asynchronous document storage.

There are three configuration options for **IDocumentContentSource** plugin. The rightmost column indicates compatibility with enabling the **IDocumentMetadataSource** plugin, shown as **IDMS** for clarity in that column heading.

Configuration name	Description	Configuration	Compatible with IDMS
Sync-only	<p>This configuration always sends document contents synchronously. The application user interface blocks the application flow for some amount of time attempting to store a recently created or uploaded document.</p> <p>If there are errors, the action that initiated the document creation or uploading will fail.</p>	<p>Connect a document content source directly in the Plugins registry for <code>IDocumentContentSource</code> in the file <code>IDocumentContentSource.gwp</code>. The main class name in that Plugins registry item must reference a <code>IDocumentContentSource</code> implementation.</p> <p>IMPORTANT For the Sync-only configuration, the class name in <code>IDocumentContentSource.gwp</code> must contain a value other than <code>gw.plugin.document.impl.AsyncDocumentContentSource</code>.</p>	Yes
Sync-first	<p>This configuration initially sends document contents synchronously. The application user interface blocks the application flow for some amount of time attempting to store a recently created or uploaded document.</p> <p>If there were errors, the document is added to a queue for sending asynchronously later, possibly from another server. All asynchronous document content sending happens only from servers with the messaging server role.</p> <p>IMPORTANT This configuration is the default in both ClaimCenter and ContactManager.</p>	<p>Connect the built-in asynchronous document content source directly in the Plugins registry for <code>IDocumentContentSource</code> in the file <code>IDocumentContentSource.gwp</code>. The built-in asynchronous document content source is the Gosu class <code>gw.plugin.document.impl.AsyncDocumentContentSource</code>.</p> <p>Additionally set the following plugin parameters:</p> <ul style="list-style-type: none"> Set the <code>SynchedContentSource</code> plugin parameter to the class that can connect to your DMS. This class must implement the <code>IDocumentContentSource</code> plugin interface. Set the plugin parameter <code>TrySynchedAddFirst</code> to <code>true</code>. 	No
Async-only	<p>This configuration always sends document contents asynchronously.</p> <p>All asynchronous document content sending happens only from servers with the messaging server role.</p>	<p>Connect the built-in asynchronous document content source directly in the Plugins registry for <code>IDocumentContentSource</code> in the file <code>IDocumentContentSource.gwp</code>. The built-in asynchronous document content source is the Gosu class <code>gw.plugin.document.impl.AsyncDocumentContentSource</code>.</p> <p>Additionally set the following plugin parameters:</p> <ul style="list-style-type: none"> Set the <code>SynchedContentSource</code> plugin parameter to the class that can connect to your DMS. This class must implement the <code>IDocumentContentSource</code> plugin interface. Set the plugin parameter <code>TrySynchedAddFirst</code> to <code>false</code>. 	No

The Sync-first and Async-only configurations are incompatible with enabling the `IDocumentMetadataSource` plugin. If you want to enable the `IDocumentMetadataSource` plugin, you must use the Sync-only configuration.

Also note that the document storage configuration that you choose affects whether ClaimCenter can enforce limitations on final documents.

The following diagram contrasts the three options for document storage asynchronous configuration.

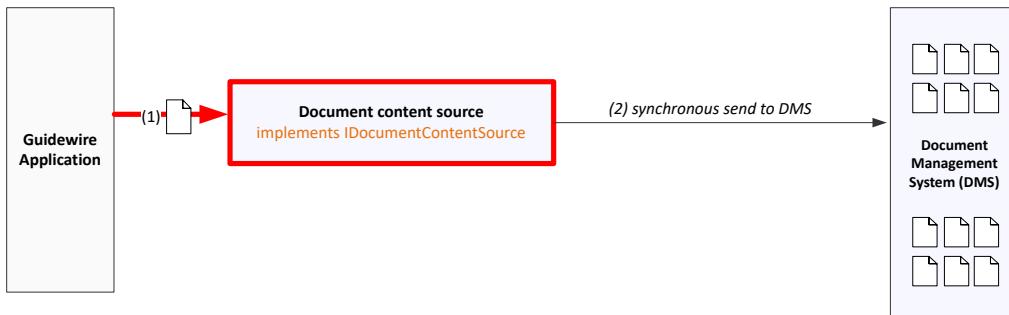
Asynchronous Document Storage Options



The thick lines and red color indicate the request from the application to the plugin implementation registered in the Plugins registry in the file `IDocumentContentSource.gwp` as the class name in the main part of the registry entry.

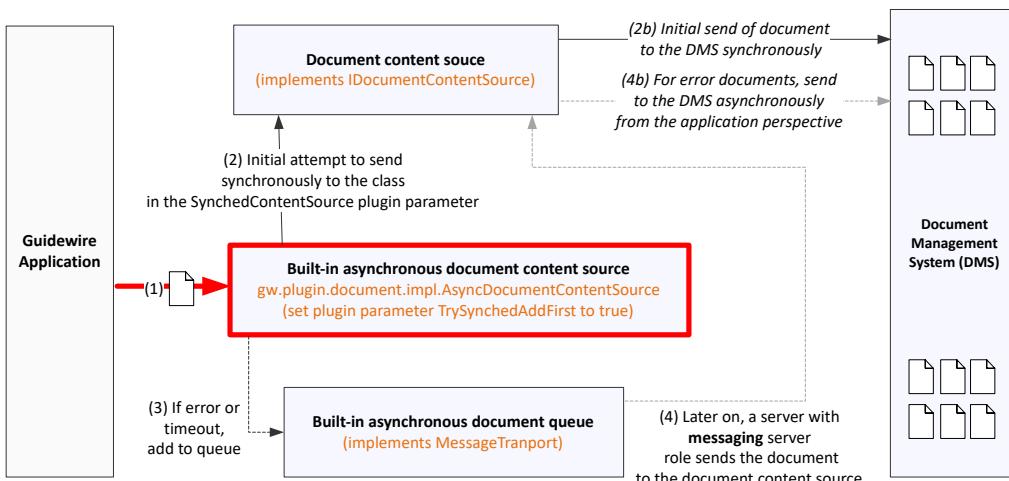
"Sync-only"

Direct synchronous connection to the `IDocumentContentSource` plugin implementation that connects to the DMS



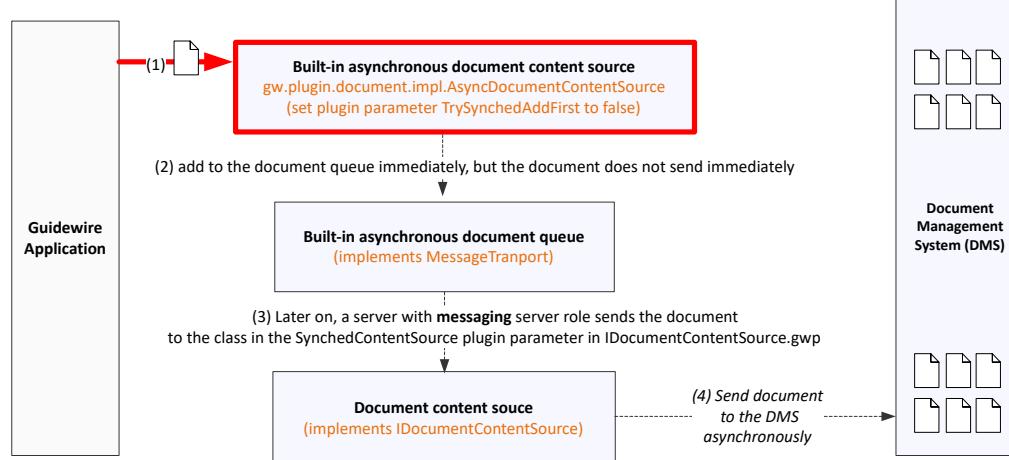
"Sync-first"

Direct connection to the built-in asynchronous content document store, which initially tries to send synchronously while the application waits. If errors occur, document is added to a queue for later sending



"Async-only"

All content goes to the built-in asynchronous document content source, which always sends asynchronously using another document content source to actually contact the DMS



Configure asynchronous document storage (sync-first or async-only)

Before you begin

In ClaimCenter and ContactManager, it is unsupported to use the asynchronous document storage plugin in conjunction with enabling the `IDocumentMetadataSource` plugin. To enable the `IDocumentMetadataSource` plugin, you must disable asynchronous document storage.

About this task

In the default configuration, both ClaimCenter and ContactManager enable the asynchronous document content storage system, but with the `TrySynchedAddFirst` plugin parameter set to `true`. This means that the initial attempt to store the document is synchronous. The application user interface blocks the application flow for some amount of time attempting to store a recently created or uploaded document. If that attempt fails, the document is added to a queue to send asynchronously.

The following procedure describes how to implement asynchronous document content storage with a custom plugin implementation of the `IDocumentContentSource` plugin.

Procedure

1. Write a content source plugin as described earlier in this topic. Your plugin implementation must implement the interface `IDocumentContentSource` and must send the document synchronously. However, do not register it directly as the plugin implementation for the `IDocumentContentSource` plugin interface in the Plugins editor in Guidewire Studio™.
2. In the Studio Project window, navigate to **Configuration > config > Plugins > registry**, and then open `IDocumentContentSource.gwp`. Studio displays the Plugins Registry editor for the default implementation of this plugin. In the default configuration, the Gosu class field has the value `gw.plugin.document.impl.AsyncDocumentContentSource`. Do not change that field.
3. In the **Parameters** table, find the `SynchedContentSource` parameter. Set it to the fully-qualified class name of the class that you wrote that implements `IDocumentContentSource`.
4. In the **Parameters** table, find the `documents.path` parameter. Set it to the local file path for temporary storage of documents waiting to be sent to document storage. In a clustered environment, map the local path to this temporary directory so that it references a server directory shared by all elements of the cluster. For example, map this to an SMB server.
5. By default, ClaimCenter first attempts synchronous sending even when asynchronous sending is enabled. It is only if that attempt fails that the document is added to the asynchronous queue. This configuration is called sync-first.

Alternatively, you can force the application to always use asynchronous sending even with initial attempts. This configuration is called async-only. To specify async-only configuration, perform the following steps.

- a) In the Studio Project window, navigate to **Configuration > config > Plugins > registry**, and then open `IDocumentContentSource.gwp`.
- b) In the **Parameters** table, find the `TrySynchedAddFirst` parameter.
- c) Set this parameter to `false`.
6. In the application `config.xml` file, set the `FinalDocumentsNotEditable` parameter to the value `false`. This allows the document status `final` to be set on documents that were sent asynchronously. Without setting this parameter to `false`, the DMS cannot add links or other meta to a final Document instance, which prevents important connections between business data entities.
7. If you are not sure if messaging destination and plugins are enabled, perform the following steps.
 - a) In the Studio Project window, navigate to **Configuration > config > Messaging**, and then open `messaging-config.xml`.
 - b) In the table, click the row for the messaging destination with name `DocumentStore`.

- c) Clear the **Disabled** check box if it is set.
- d) In the Studio Project window, navigate to **Configuration > config > Plugins > registry**, and then open `documentStoreTransport.gwp`.
- e) Clear the **Disabled** check box if it is selected.

Disable asynchronous document content storage completely (sync-only)

About this task

If you disable the asynchronous document content storage plugin, document contents always send synchronously. This is known as the sync-only configuration.

Procedure

1. In the Studio Project window, navigate to **Configuration > config > Plugins > registry**, and then open `IDocumentContentSource.gwp`.
2. Set the class name to the fully-qualified class name of your own `IDocumentContentSource` implementation or the built-in `IDocumentContentSource` implementation you want to use. The class name may already be in the plugin parameter `SynchedContentSourced`. You can copy that value and remove the `SynchedContentSourced` plugin parameter.

Be sure that the plugin registry does not set the class to be the internal class `gw.plugin.document.impl.AsyncDocumentContentSource`. To disable asynchronous document content storage, the Plugins registry for `IDocumentContentSource.gwp` must reference a `IDocumentContentSource` implementation other than that class.

The default Plugins registry item is configured to a reference a Gosu class. If your `IDocumentContentSource` implementation class was not implemented in Gosu, you must remove and then re-add the Plugins registry item for that plugin interface. Click the red minus sign to remove the Gosu plugin configuration. If the plugin implementation is in Java but without using OSGi, click the green plus sign and then click **Java**. If the plugin implementation is an OSGi bundle, click the green plus sign and then click **OSGi**.

3. In the application `config.xml` file, set the `FinalDocumentsNotEditable` parameter to the value `true`. Setting it to `true` enforces the document status `final`. The value `true` is incompatible with documents that were sent asynchronously. When disabling asynchronous document content storage, it is best to restore enforcement of the document status `final`.
4. Disable the messaging destination by performing the following steps.
 - a) In the Studio Project window, navigate to **Configuration > config > Messaging**, and then open `messaging-config.xml`.
 - b) In the table, click the row for the messaging destination with name `DocumentStore`.
 - c) Select the **Disabled** checkbox.
 - d) In the Studio Project window, navigate to **Configuration > config > Plugins > registry**, and then open `documentStoreTransport.gwp`.
 - e) Select the **Disabled** checkbox.

Errors and document validation in asynchronous document storage

If you use asynchronous document storage at all (either Sync-first or Async-only configuration), document content storage errors occur at storage time. Document storage time may be much later than the time of uploading or creating the document.

When the asynchronous document content storage code fails to store in the database, ClaimCenter triggers a `FailedDocumentStore` event on the relevant `Document` entity instance. The event is important because it is the only notification that document storage failed. Use this event to create a notification for administrators or users.

To handle a notification, you need to do the following operations.

1. Create a messaging destination that listens for the `FailedDocumentStore` event.
2. Create Event Fired rules that handle this event for this destination.

The following suggestions present possible methods to handle the problem.

- Email the creator of the `Document` entity.
- Create an activity on the primary object or root object.

Optional document validation for asynchronous document storage

For your DMS, it is possible that there are predictable circumstances that generate errors on storage, such as invalid file types. If you can write Gosu code that defines the error conditions, you can modify the application (ClaimCenter or ContactManager) to validate the document at the time of creation or uploading. With document validation, the user who uploads or creates an invalid document gets immediate feedback of the problem.

In Studio, edit the class that implements the asynchronous document content source, `AsyncDocumentContentSource`. Edit the `addDocument` method, which is called immediately upon document creation or uploading. By default, this method does no validation. You can modify this method to add a call to your own code that performs validation. Add your call to your validation code immediately before the method call to `createTemporaryStore`.

Your validation code must throw a `UserDisplayableException` exception if validation fails.

Final documents

ClaimCenter and ContactManager have optional support for final documents, which are documents that are not allowed to be modified after completion. Final document are implemented as documents whose `Document.Status` property has the value `DocumentStatusType.TC_FINAL`.

The behavior and implementation details for final documents are different depending on whether you want to enable the `IDocumentMetadataSource` plugin. If that plugin is enabled, the application stores document metadata in an external DMS rather than the internal database.

Final documents with `IDocumentMetadataSource` enabled

In the default configuration, if the `IDocumentMetadataSource` plugin is enabled, the application does not have a special behavior for final documents. A final document can be edited and can be replaced by a new uploaded file.

With the `IDocumentMetadataSource` plugin enabled, the metadata for a document is stored in an external DMS. Each time ClaimCenter receives a document from the DMS, ClaimCenter creates a new `Document` entity in memory to hold the metadata temporarily. The built-in permissions check for final documents uses the value of the `Status` property of the `Document`. However, for newly created entity instances, the `Status` property is in an initial state that does not mirror the status within the DMS. Therefore, in the default configuration, the code does not prohibit editing or uploading new versions.

Set the `config.xml` property `FinalDocumentsNotEditable` to `false`.

In the default configuration, the `config.xml` file for ContactManager sets `FinalDocumentsNotEditable` to `false`. In contrast, ClaimCenter does not explicitly define the property, which has the effect of the default value `true`.

Final documents with `IDocumentMetadataSource` disabled

If the `IDocumentMetadataSource` plugin is disabled, the behavior depends on how you configure asynchronous document storage and the value of the `config.xml` property `FinalDocumentsNotEditable`.

If you use either the Sync-first or Async-only configurations, you must set `FinalDocumentsNotEditable` to `false`. This forces the application to not have a special behavior for final documents. A final document can be edited and can be replaced by a new uploaded file. This is the only supported option for Sync-first or Async-only configurations for document storage due to the interaction with asynchronous document storage.

If you are using the Sync-only configuration, you can configure application behavior with the `config.xml` property `FinalDocumentsNotEditable`.

- If `FinalDocumentsNotEditable` is `true`, the ClaimCenter user interface prevents metadata changes as well as uploading a new version of the document to the DMS.

The ContactManager interface is slightly different. If `FinalDocumentsNotEditable` is `true`, the ContactManager user interface prevents uploading of documents but not metadata changes.

- If `FinalDocumentsNotEditable` is `false`, then metadata is editable and you can upload a new version of the document to the DMS.

If it is important for final documents that the application prohibit either the metadata changes or the upload button, but not both, this is customizable. Set `FinalDocumentsNotEditable` to `false`, then modify the `DocumentDetailsPopup.pcf` file to change the logic of the **Editor** or **Update** button.

In the default configuration, the `config.xml` file for ContactManager sets `FinalDocumentsNotEditable` to `false`. In contrast, ClaimCenter does not explicitly define the property, which has the effect of the default value `true`.

Document interactions also depend on user permissions

There are document behaviors that are allowed or prohibited based on user permissions, independent of the `IDocumentMetadataSource` configuration. For example, if a user does not have the permission to edit documents, the **Edit** button for those screens is never enabled.

Be aware that different Guidewire applications have different behaviors with respect to how user permissions affect final documents. In ClaimCenter, a privileged user can delete final documents. In ContactManager, no users can delete final documents.

APIs to attach documents to business objects

Gosu APIs to attach documents to business objects

ClaimCenter provides document-related Gosu APIs. Your business rules can add new documents to certain entity types.

First, instantiate and initialize a new `Document` entity instance.

For ClaimCenter, to attach a new `Document` object to a `Claim` or `Exposure` object, call the `addToDocuments` method and pass the method a new `Document` entity instance. You can also affiliate a document with a `Matter` or `ClaimContact` object but there is no corresponding `addToDocuments` method on those entities. However, you can set the `Document.Matter` or `Document.ClaimContact` property to create the link. Note that a document can be associated with no more than one of the following: a matter, an exposure, or a claim contact.

You can associate a document with multiple objects (a many-to-many relationship) with entity types `CheckSet`, `ReserveSet`, and `Activity`. The `addToDocuments` method for those entity types do not take a `Document` entity argument. Instead, they take a separate entity type that helps link entities in the many-to-many relationship. The `CheckSet` and `ReserveSet` versions of the method take a `TransactionSet` entity instance. The `Activity` version of the method takes an `ActivityDocument` entity instance.

For ContactManager, to attach a new `Document` object to an `ABContact` object, call the `addToDocuments` method and pass the method a new `Document` entity instance.

Web service APIs to attach documents to business objects

ClaimCenter provides a web services API for linking business objects (such as a claim) to a document. This allows external process and document management systems to work together to inform ClaimCenter after creation of new documents related to a business object. For example, in paperless operations, new postal mail might come into a scanning center to be scanned. Some system scans the paper, identifies with a business object, and then loads it into an electronic repository.

The repository can notify ClaimCenter of the new document and create a new ClaimCenter activity to review the new document. Similarly, after sending outbound correspondence such as an email or fax, ClaimCenter can add a reference to the new document. After the external document repository saves the document and assigns an identifier to retrieve it if needed, ClaimCenter stores that document ID.

The `ClaimAPI` web service includes a method to add documents.

- `createDocument` – Add a new document to an existing claim using a `DocumentDTO` object to encapsulate the important fields, including `DocumentDTO.ClaimPublicID` for the claim's public ID.

See also

- “Adding a document from external systems” on page 156

Document management servlet for testing vendor documents

ContactManager includes a demonstration document management system implemented as a servlet. Because this code is intended for testing vendor documents in ContactManager, this code is implemented only in ContactManager not ClaimCenter. The ContactManager servlet is called `DMSServlet`, which defines a RESTful API for use in testing a simulated DMS that returns URLs to vendor documents.

Never use the `DMSServlet` code as a document content source in any production system. It is intended for testing and performing demonstrations of the ClaimCenter vendor documents feature.

In ContactManager, use the related document plugin implementation classes to simulate either a content store (with no metadata) or a complete DMS (content and metadata). The ContactManager application is configured by default to use the `DMSServlet` as the DMS for content only. You can enable the `DMSServlet` for metadata also, but if you must disable asynchronous document content storage.

You can configure the behavior using system properties within the ContactManager application only.

If you want to simulate a DMS for testing ClaimCenter documents without a separate DMS, use the document content source plugin implementation in ClaimCenter that save documents and metadata locally. Those implementation classes are available in both ClaimCenter and ContactManager.

DMS servlet plugins and ContactManager

The ContactManager application is configured by default to use the `DMSServlet` as the DMS for both content and metadata. This configuration is implemented by plugin implementation classes that are registered with the correct plugin web service collections in the Plugin Registry editor.

This topic includes the following:

- “Servlet-backed base class” on page 219
- “Servlet-backed document content source” on page 220
- “Servlet-backed document metadata source” on page 220
- “Document search criteria” on page 220

Servlet-backed base class

The `ServletBackedDocumentBaseSource` class is the base class for the classes that implement both the document content source and the document metadata source.

This class reads the following plugin parameters from the Plugins registry in ContactManager.

DMS

If present, it must point to the servlet URL, and can use shortcuts defined in `suite-config.xml`, such as `"{px}/service/dms/"`. This value can be overridden by system property `gw.document.DMSServlet.url-root`, which can be set in a GUnit test to change the URL.

Username

User name for a user in ContactManager.

Password

Password for that user in ContactManager.

PingThrottleMinutes

For outbound connections, ensure outgoing connections for at least this many minutes. If there has been idle time, the plugin sends a ping to the server after this many minutes.

Servlet-backed document content source

The servlet-backed implementation of `IDocumentContentSource` can be used either with `IDocumentMetadataSource` enabled or disabled. The `IDocumentContentSource` plugin interface sends and receives document content from the DMS. If `IDocumentMetadataSource` plugin is disabled in the ContactManager Studio registry, the metadata is stored internally in the local ContactManager database.

In the default configuration, the ContactManager Studio application registers the implementation class `gw.plugin.document.impl.AsyncDocumentContentSource`, which implements asynchronous document storage.

In the Plugins registry for that plugin interface in ContactManager, there is a plugin parameter called `SynchedContentSource` with value `gw.plugin.document.impl.ServletBackedDocumentContentSource`. That plugin parameter tells the asynchronous document storage system which plugin implementation actually handles the document storage. In this case, the `SynchedContentSource` plugin parameter specifies the built-in class that implements communication with the servlet.

To facilitate multiple application communication, such as ClaimCenter-ContactManager communication, this document content source always uses the URL response type for document contents.

Servlet-backed document metadata source

In ContactManager, the servlet-backed implementation of `IDocumentMetadataSource` exists but is optional and not enabled in the default configuration. Because it is not enabled in the Plugins registry, document metadata is stored internally. The `IDocumentMetadataSource` plugin interface sends and receives document metadata in the DMS.

To enable the servlet-backed document metadata source in ContactManager, first open ContactManager Studio. Navigate to **configuration > config > Plugins > registry** and click `IDocumentMetaDataSource.gwp`. Uncheck the **Disabled** checkbox.

In ContactManager, enabling asynchronous document content storage is not supported if you enable the document metadata source (`IDocumentMetadataSource`) plugin. If you enable `IDocumentMetadataSource` with any plugin implementation (whether for `DMSServlet` or another one), you must disable asynchronous document storage.

Document search criteria

For document search, each search can specify whether query includes pending documents, which may exist if you use asynchronous document content storage. Pending documents are documents that are added to the database but not yet sent by the asynchronous document content source to the synchronous document content source.

The document search behavior is different for `IDocumentMetadataSource` enabled (a plugin handles metadata) compared to disabled (the metadata is in the local database).

With `IDocumentMetadataSource` plugin disabled in the Plugins registry, the local application database contains the document metadata. With all documents in the local application database, the search contains both pending and permanent documents. If `IDocumentMetadataSource` is enabled, by default only permanent documents are returned from the search.

The `Pending` attribute on the search criteria object allows you to be explicit with the behavior.

- Set `Pending` to `true` to return only pending documents.
- Set `Pending` to `false` to return only permanent documents.
- Set `Pending` to `null` (the default) to return all documents both pending and permanent.

Document DTO used by the DMS servlet

To send and receive document metadata between the servlet and its clients, the DMS servlet and associated code uses a data transfer object (DTO) called the <document> XML element. This XML element is defined in the `DocumentDTO.xsd` file in the `gw.document` package. Search results include <documents> element that is a sequence of <document> elements. These XML objects are used only by the document management code associated with the `DMSServlet` REST APIs, which are intended for demonstration of a simulated document management system (DMS).

IMPORTANT: The DTOs for the DMS servlet in the `DocumentDTO.xsd` file are different from the similarly named objects defined in Gosu classes and used by general purpose WS-I web services. For example, the ClaimCenter Gosu class `gw.webservice.cc.cc1000.dto.DocumentDTO` is used by WS-I web services to represent a document.

Configuring the DMS servlet in ContactManager

You can configure the behavior with the following system properties in ContactManager. Some properties can optionally be set by plugin parameters also. If you define a plugin implementation for `IDocumentMetadataSource`, the values for those plugin parameters must be the same for the `IDocumentContentSource` and the `IDocumentMetadataSource` plugins in the Plugins registry.

Configuration system property	Description
<code>gw.document.DMSServlet.url-root</code>	The root of any returned URL.
<code>gw.document.DMSServlet.root-dir</code>	The root of the directory for all DMS files. The default is <code>/tmp/dms</code> .
<code>gw.document.DMSServlet.db-url</code>	The external URL for an external H2 database to use instead of the in-memory server. The default is empty, which means to use the in-memory database.
<code>gw.document.DMSServlet.db-username</code>	Database user name. Defaults to <code>sa</code> .
<code>gw.document.DMSServlet.db-password</code>	Database password. Defaults to <code>sa</code> .
<code>gw.document.DMSServlet.props</code>	A comma-delimited list of properties that clients can use in searches. The servlet populates this property in the internal metadata database even before any document is saved.

Except for the `db-username` and `db-password` properties, the other `DMSServlet` properties can be optionally set as a plugin parameter to the `IDocumentContentSource` in the Plugins registry in ContactManager. In other words, in ContactManager Studio, in the Plugins registry, edit `IDocumentContentSource.gwp` and add this property as a plugin parameter. If you are using the DMS servlet for metadata, also edit the file `IDocumentMetadataSource.gwp` and add this property as a plugin parameter.

If this parameter is set as a system property, the system property overrides the configuration property in the plugin parameters in the Plugins registry.

REST APIs of the DMS servlet

The ContactManager DMS servlet accepts the following REST API requests. The rightmost column indicates whether the servlet authenticates the request with the user name and password that are defined in the system properties. Authentication happens with standard HTTP BASIC authentication.

HTTP verb	REST request URL	Description	Auth
GET	clean	Cleans the database if the database is local.	No

HTTP verb	REST request URL	Description	Auth
GET	sleep?minutes=<n>	For testing delays and failures, forces a delay for n minutes before responding to the request. Note that if n minutes is larger than the timeout interval, a request fails.	No
GET	addLinkProperty? Document.PublicID=<docid> &JoinTable.Type=<table> &JoinTable. JoinedProperty=<attribute> &Joined.PublicID=<joinedid>	<p>Adds a property for the joined bean. Parameters are:</p> <ul style="list-style-type: none"> • docid – The document public id • table – the join table name • attribute – the joined entity's reference in the join table • joinedid – the joined bean's public ID <p>The order of parameters is irrelevant.</p> <p>The request returns an error if the properties are not found or the link was not added.</p>	No
GET	removeLinkProperty? Document.PublicID=<docid> &JoinTable.Type=<table> &JoinTable. JoinedProperty=<attribute> &Joined.PublicID=<joinedid>	<p>Removes the property for the joined bean. Properties are:</p> <ul style="list-style-type: none"> • docId – The document public id • table – the join table name • attribute – the joined entity's reference in the join table • joinedid – the joined bean's public ID <p>The order of parameters is irrelevant.</p> <p>The request returns an error if the properties are not found or the link was not added.</p>	No
GET	isLinkProperty? Document.PublicID=<docid> &JoinTable.Type=<table> &JoinTable. JoinedProperty=<attribute> &Joined.PublicID=<joinedid>	<p>Checks if there is a property for the joined bean. Parameters are:</p> <ul style="list-style-type: none"> • docId – The document public id • table – The join table name • attribute – The joined entity's reference in the join table • joinedid – the joined bean's public ID <p>The order of parameters is irrelevant.</p> <p>Returns true if there is a link or false if there is not a link. The request returns an error if properties are not found.</p>	No
GET	findLinkProperty? Document.PublicID=<docid> &JoinTable.Type=<table> &JoinTable. JoinedProperty=<attribute>	<p>Returns the public IDs of all joined beans of a particular type. Parameters are:</p> <ul style="list-style-type: none"> • docId – The document public ID • table – The join table name • attribute – the joined entity's reference in the join table <p>The order of parameters is irrelevant.</p> <p>Returns a list of public IDs, which one ID on each line. The request returns an error if the properties are not found.</p>	No
GET	ping	Returns a text/plain empty document. This is intended for testing to confirm that the server is running.	No
GET	metadata	Gets the template document XML element.	No
GET	content/<publicid>/<filename>	<p>Returns the content for the document.</p> <p>IMPORTANT: The MIME type of the returned data is dependent on the MIME type of the content of this specific document.</p>	No

HTTP verb	REST request URL	Description	Auth
GET	metadata?<searchcriteria>	Returns a set of matching documents for search criteria, returned as a Documents XML object. In addition to the attributes discoverable in the template, you can use the following attributes as parameters in the search criteria. <ul style="list-style-type: none">• _Author – case insensitive match on author• _NameOrID – case insensitive match on name or on document UID The following attributes are written by the document management plugin implementations but do currently affect actual search results: _GetNumResultsOnly, _IncludeTotal, _MaxResults, _StartRow, _SortColumns.	No
GET	metadata/<publicid>	Gets the document metadata as an XML element	No
PUT	metadata	Creates a new document XML element as from an input stream. Creates a new entry if needed for that public ID or assigns a new public ID to the document. This request returns the public ID as text in the text/plain MIME type.	Yes
PUT	metadata/<publicid>?update	Updates an existing document with a document XML element as input stream. If the public ID is null, creates a new document. This request returns the public ID as text in the text/plain MIME type.	Yes
PUT	content/<publicid>/<filename>	Uploads the content for the document based on the input stream.	Yes
DELETE	metadata/<publicid>	Deletes the document.	Yes

Related APIs for DMS servlet

There are APIs that exist to support the DMS servlet system. If you are testing the DMS servlet or related code, feel free to use them. Remember that the DMS servlet system is for testing use only. Never use the DMS servlet in a production system.

Method: addToXml

All Guidewire applications have a Document entity enhancement method called `addToXml`. The method converts the Document entity metadata to the XML format necessary for the DMS servlet.

Enhancement methods (ContactManager only)

In ContactManager, there are additional enhancement methods on the Document entity instance implemented by the GWABDocumentEnhancement enhancement.

- `startNewList` – adds a new list of documents in the format used by the DMS server.
- `addToList` – add the current Document entity instance to a list of documents.
- `asXml` – converts the current Document entity instance to XML.
- `removeJoinTableEntries` – Remove join table entries for linking to other documents.

DocumentsUtilBase methods (ClaimCenter and ContactManager only)

The `gw.documents.DocumentsUtilBase` class has static utility methods that you can use, but you cannot edit the source code, which is written in Java.

- `getExcludeFieldsFromXml` – Determine what fields to exclude from XML.
- `getFileExtensionForDocument` – Get the file name extension base on the document MIME type.

- `getFullFilename` – Get the file name with extension for a document.

The following `DocumentsUtilBase` static methods are implemented as Gosu enhancement methods in the file `GWDocumentsUtilBaseEnhancement.gsx`.

- `fetchAndUpdate` – Find or create the document based on the DTO type. The method populates the properties if the bean is nor or update is set.
- `xmlTemplate` – Return an XML document with all attributes set to acceptable default. It is primarily used to see what parameters can be passed to the search function.

Document production

ClaimCenter provides a user interface and APIs to create documents, download documents, and produce automated form letters. This topic discusses creating new documents from templates.

ClaimCenter can create new documents from forms, form letters, or other structured data. For example, notification emails, letters, or faxes created by business rules to an insured customer. The resulting new document optionally can be attached to business data objects. ClaimCenter can create some types of new documents from a server-stored template without user intervention.

In contrast to ClaimCenter, the ContactManager application supports documents attached to vendor contacts only. This is part of the integration with ClaimCenter, and not used in direct integrations with PolicyCenter or BillingCenter. For use with ClaimCenter, the ContactManager application supports document upload for vendor contact documents but does not support any document production.

In ClaimCenter, users can create new documents from a template and then attach them to a claim or other ClaimCenter object. This is useful for generating forms or form letters or any type of structured data in any file format. Or, in some cases ClaimCenter creates documents from a server-stored template without user intervention. For example, ClaimCenter creates a PDF document of an outgoing notification email and attaches it to the claim.

To support document production, you typically design a large number of templates that support specific business needs by using the built-in types of document production types.

For every document template, you must create two files.

A document template source file

The source file of the appropriate type, such as a Microsoft Word document for a Microsoft Word template. The file must be specially prepared according to its type such that form fields are inserted in the correct places. Microsoft Word files must have form fields configured from an appropriate data source. Microsoft Excel documents must have named regions. PDF files must have form fields. Gosu templates must have embedded code that uses the appropriate symbol names.

A document template descriptor file

The document template descriptor describes metadata about the template and how to insert parameters into the template. The document template descriptor describes an actual document template, such as a Microsoft Word mail merge template, an Adobe PDF form, or a Gosu template.

The amount of configuration you need depends on how complex your templates are. For example, if you have many parameters or need embedded Gosu code, configuration is more complex.

Typical requirements are for document production with document templates and using the built-in document production types. Only in rare cases do you need to create new document production implementations. In these cases, you can create new document production types by writing an implementation of the `IDocumentProduction` plugin.

Document production types

Each application uses a different mechanism for defining fields to be filled in by the document production plugins. The following table describes the document production plugins in the base configuration and their associated formats.

Application or format	Description
Microsoft Word	<p>This document production type uses Microsoft Word mail merge functionality to generate a new document from a template and a descriptor file. The template must use mail merge fields to define the field names. One way to define the field names is to use a CSV file with field names in the headings.</p> <p>For an example, ClaimCenter includes a Word document called “Reservation of Rights” and an associated CSV file. That CSV file is present so that you can manually open the Word template and see how form fields correspond to fields defined in the CSV file. ClaimCenter does not require or use this document.</p> <p>Guidewire recommends that you use Word format instead of RTF for document production.</p>
Adobe Acrobat (PDF)	PDF document production relies on form fields defined in the Acrobat file. PDF document creation requires a license.
Microsoft Excel	This document production type uses Microsoft Excel native named fields functionality to define the form fields. The named fields in the Excel spreadsheet must match the FormField names in the template descriptor. After the merge, the values of the named cells become the values extracted from the descriptor file.
Gosu template	<p>Gosu templates can generate any kind of text file, including plain text, RTF, CSV, HTML, XML, or any other text-based format.</p> <p>The document production plugin retrieves the template and uses Gosu to populate the document from Gosu code and values defined in the template descriptor file. Based on the file’s MIME type and the local computer’s settings, the system opens the resulting document in the appropriate application.</p>

Understanding synchronous and asynchronous document production

The document production (`IDocumentProduction`) plugin interface manages creation of new documents. Each `IDocumentProduction` implementation provides two basic modes of document production.

- Synchronous production – The document contents are returned immediately from the creation methods. All built-in document production types support synchronous production.
- Asynchronous production – The creation method returns immediately, but the actual creation of the document is performed in a different thread and the document may not exist for some time. All of the built-in document production types support asynchronous production, but asynchronous document production is never called from the default user interface or business rules.

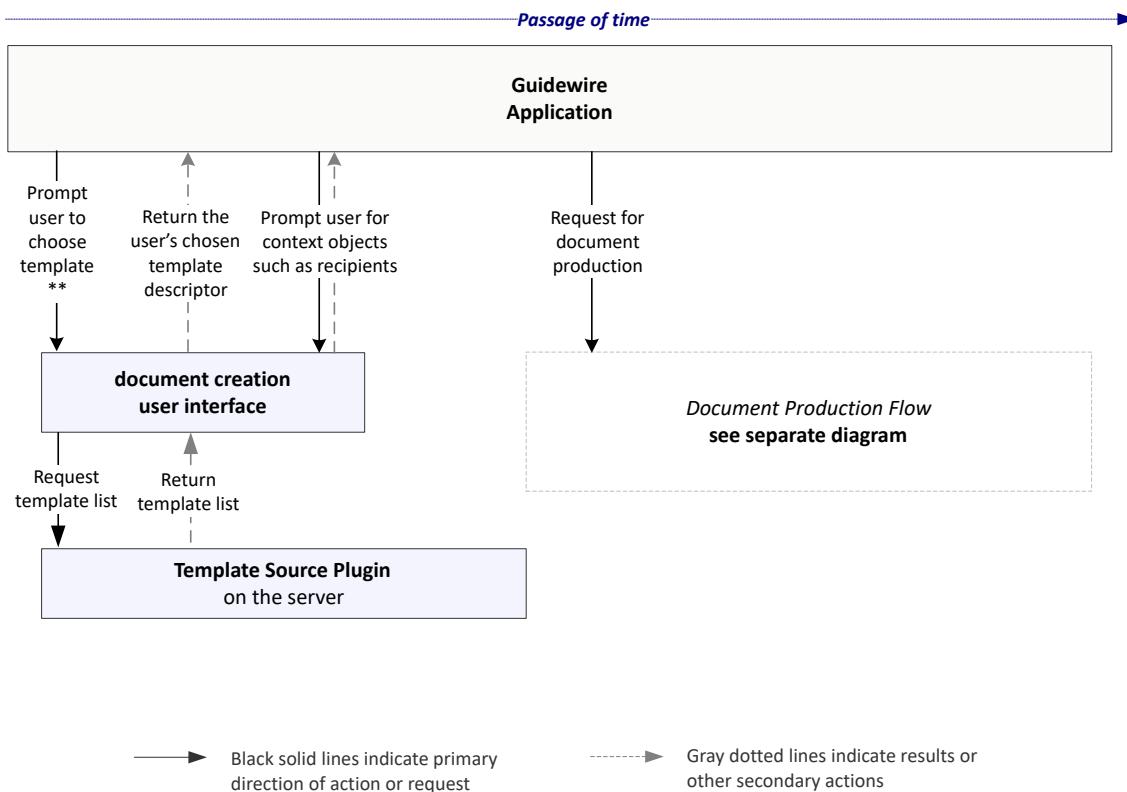
After document production, storage plugins store the document in the document management system.

User interface flow for document production

From the ClaimCenter user interface, create forms and letters and choose the desired template. You can select other parameters (some optional) to pass to the document production engine. ClaimCenter supports Gosu-initiated automatic document creation from business rules. The automatic and manual processes are similar, but the automatic document creation skips the first few steps of the process. The automated creation process skips some steps relating to parameters that the user chooses in the user interface. Instead, the Gosu code that requests the new document sets these values.

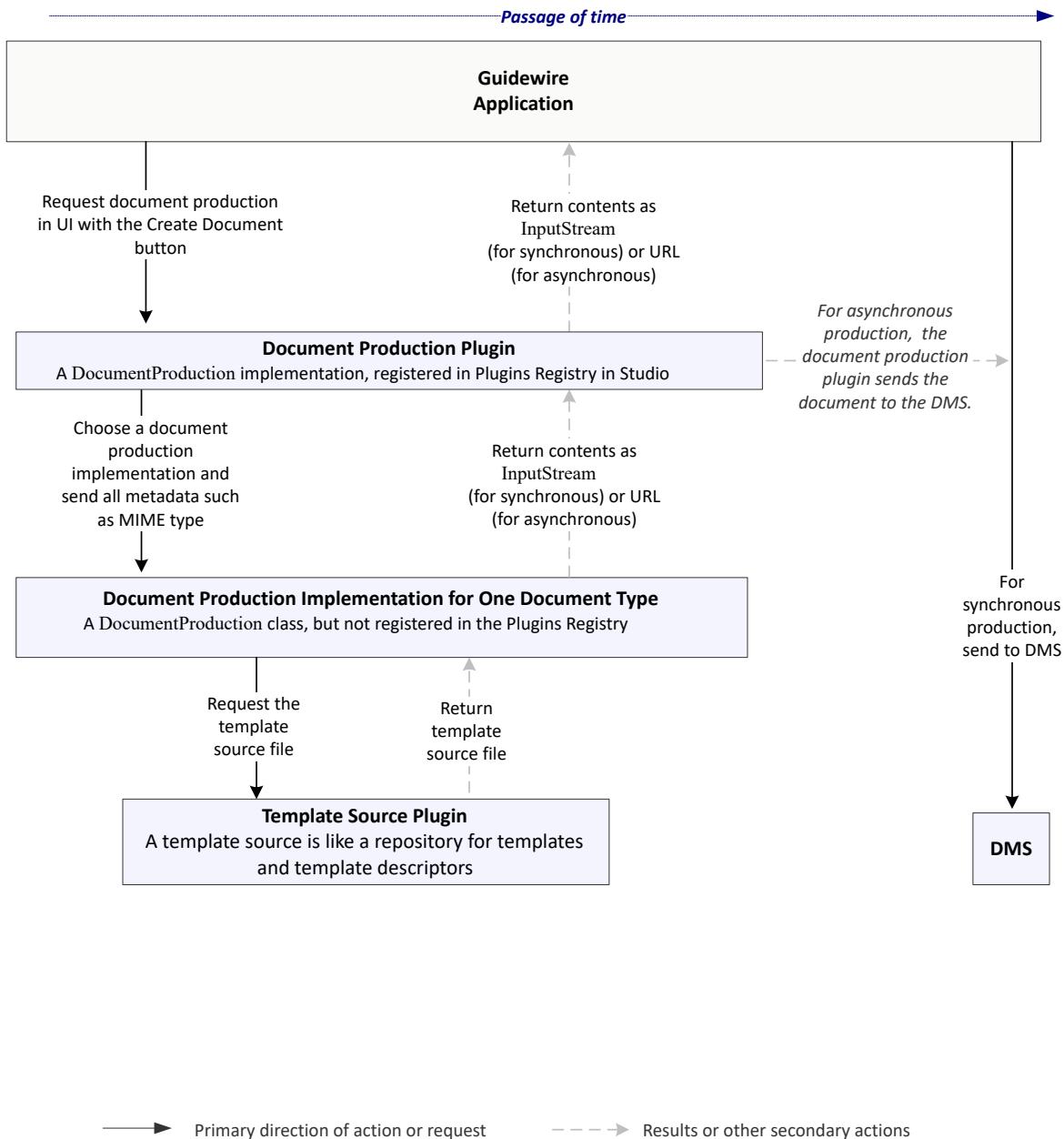
The following diagrams illustrate interactions between ClaimCenter, the various plugin implementations, and an external document management system during document generation.

Interactive Document Creation UI Flow



** for document production from activities, the activity defines the template in its DocumentTemplate property.
For document creation from Gosu business rules, you can specify the template explicitly.

Document Production Flow



The chronological steps are described below.

1. ClaimCenter invokes the document creation user interface to let you select a form or letter template.
2. The ClaimCenter document creation user interface requests a list of templates from the *template source*, which is a plugin that acts like a repository of templates and their descriptors (metadata).
3. The template source returns a list of potential templates.
4. The document creation user interface displays a list of available templates for this user interface context, based on the programming symbols used on the page and the template's required symbols. The user interface lets you choose one after optional searching or filtering a list of potential templates. Searching and filtering uses template

descriptors, which contain template metadata. Template metadata is information about the templates themselves. For example, the template specifies a list of required symbols that indicate required data that must be available in that user interface context. The template descriptor optionally can identify additional localized versions of the template file for a user to select.

5. After you select a template, the document creation user interface returns the template descriptor information for the chosen template to ClaimCenter.
6. ClaimCenter prompts you through the document creation user interface for other document production information called context objects for the document merge. For example, choose the recipient of a letter or other required or optional properties for the template. After you enter this data, the user interface returns the context object data to ClaimCenter.
7. ClaimCenter requests document production from the document production (`IDocumentProduction`) plugin implementation that is registered in the Plugins registry in Studio. The document production plugin gets a template descriptor and a list of parameter data values. In the default implementation, the built-in document production plugin implementation is merely a dispatch mechanism that maps from a MIME type to another `IDocumentProduction` plugin implementation. Only the main document production (`IDocumentProduction`) dispatcher is explicitly registered in the Plugins registry but other `IDocumentProduction` implementations do the actual document production work.
8. The document production plugin appropriate for that MIME type uses the template descriptor to request the actual template file from the template source. In the default configuration, the document source is a directory on the server that holds the files. If the user requests a localized version of the template, the template source returns the localized template.
9. The document production plugin is responsible for generating a new document, typically by merging a template file with parameter data. ClaimCenter includes document production plugins that process Microsoft Word files, Adobe Acrobat (PDF) files, Microsoft Excel files, and Gosu templates. You can also generate text formats such as HTML and CSV using Gosu templates. If the document production request was triggered by Gosu rules, the application populates any default values that are not explicitly provided.

The result of this step is a merged document.
10. After production, the document is added to the document management system. The details vary depending on which APIs created the document, the application/code context, and how you configure the application.
 - For synchronous document production from an activity or Gosu rules, the application automatically sends the document to the document management system.
 - For synchronous document production from the application user interface, the user must click the **Update** button to save the document and send it to the document management system.
 - For asynchronous document production APIs, you must modify the `IDocumentProduction` plugin to support direct interaction with your document management system after production is complete.
 - If document production happens on an external system, that system can add documents to the document management system. Additionally the external system can link the document to ClaimCenter business objects with ClaimCenter web services APIs.

The preceding steps describe the base implementation. You can configure many parts of the user interface and underlying plugins to support your own business needs.

Document production plugins

The `IDocumentProduction` plugin is the main interface to a document creation system for a certain type of document. The document creation process may involve extended workflow or asynchronous processes.

Typical requirements are for document production with document templates and using the built-in document production types. Only in rare cases do you need to create new document production implementations. In these cases, you can create new document production types by writing an implementation of the `IDocumentProduction` plugin.

It is critical to understand that there are two types of `IDocumentProduction` plugin implementations.

- The built-in dispatcher implementation – In the Plugins registry in Studio, the registered `IDocumentProduction` plugin implementation is `gw.plugin.document.impl.LocalDocumentProductionDispatcher`. Its job is to determine what other `IDocumentProduction` plugin implementation can do the actual work for that document. For typical use, continue to register the existing implementation in the Plugins registry. However, you can configure its dispatching algorithm. Modify the plugin parameters in the `IDocumentProduction.gwp` registry item to map a MIME type to a `IDocumentProduction` plugin implementation class that can handle that MIME type.
- Document production plugin implementations for a specific document type – Other `IDocumentProduction` plugin implementations are called by the built-in dispatcher but are not registered in the Plugins Registry in Studio as unique .gwp files. For example, there is a `IDocumentProduction` plugin implementation that can handle Microsoft Word production. However, it is not independently registered as a separate item in the Plugins registry. Instead, the plugin parameters in the `IDocumentProduction.gwp` registry item maps a MIME type to a `IDocumentProduction` plugin implementation that is specific to a document type.

When the main document production plugin implementation gets a request for a new document, the main document production plugin implementation dispatches the request to the appropriate `IDocumentProduction` plugin implementation. The MIME type of the document determines which `IDocumentProduction` implementation to use.

ClaimCenter supports two types of document production. In synchronous production, the document contents are returned immediately. In asynchronous production, the creation method returns immediately, but creation happens later. Each type of document production has different integration requirements.

For synchronous production, ClaimCenter and its installed document storage plugins are responsible for persisting the resulting document, including both the document metadata and document contents. In contrast, for asynchronous document creation, the document production (`IDocumentProduction`) plugins are responsible for persisting the data.

The `IDocumentProduction` plugin has two main methods.

- `createDocumentSynchronously` – Creates a document synchronously.
- `createDocumentAsynchronously` – Creates a document asynchronously.

There are additional interfaces that assist the `IDocumentProduction` plugin.

- Template source and template descriptor interfaces – The interfaces `IDocumentTemplateDescriptor` and `IDocumentTemplateSource` encapsulate the basic interface for searching and retrieving the templates that describe a document to create. The descriptive information includes the basic metadata (name, MIME type, and so on) and a pointer to the template content on the file system. The `IDocumentTemplateDescriptor` implementation describes the template that is used to create documents. The built-in `IDocumentTemplateSource` implementation searches all available templates and retrieves document templates from the file system.

The built-in implementations of `IDocumentTemplateSource` and `IDocumentTemplateDescriptor` are sufficient for nearly all requirements. Typically there is no need to re-implement these plugin interfaces. You can define your template descriptors using the XML file format used by the built-in code that implements these plugin interfaces.

- Interfaces for built-in production types – For the server-side production of PDF files, Microsoft Excel files, and Microsoft Word files, ClaimCenter provides interfaces that define the core functionality. In the base configuration, ClaimCenter provides implementations in proprietary code written in Java. You can replace the built-in implementation with an entirely new implementation class, but the built-in implementations support most requirements. If you write your own version, edit the appropriate plugin parameter for the `IDocumentProduction` plugin interface in the Plugins Registry in Studio. Set the value for that MIME type to the new implementation class. This parameter applies to the following interfaces.
 - `IPDFMergeHandler` – Creates a merged PDF document.
 - `ServerSideWordDocumentProduction` – Creates a merged Microsoft Word document.
 - `ServerSideExcelDocumentProduction` – Creates a merged Microsoft Excel document.

Implementing synchronous document production

The `createDocumentSynchronously` method returns a `DocumentContentsInfo` object. The `ResponseType` property specifies the format of the retrieved document contents. Defined response type values are `DOCUMENT_CONTENTS`, `URL`,

and URL_DIRECT, although the only applicable value when creating a document is DOCUMENT_CONTENTS. Each value is referenced through `gw.document.DocumentContentsInfo.ContentResponseType`, as shown in the example below.

```
returnedDocContentsInfo.ResponseType =  
    gw.document.DocumentContentsInfo.ContentResponseType.DOCUMENT_CONTENTS
```

The response types are described below.

- DOCUMENT_CONTENTS – The `DocumentContentsInfo.InputStream` property references an input stream that contains the document contents as a stream of bytes.
- URL, URL_DIRECT – Not applicable in this context.

To persist the document, the caller of the `createDocumentSynchronously` method must pass it to the `addDocument` method of the `IDocumentContentSource` plugin.

See also

- “Implementing a document content source for external DMS” on page 200

Implementing asynchronous document production

Asynchronous document creation returns immediately, but the actual creation of the document is performed in a different thread and the document may not exist for some time. The `IDocumentProduction` plugin method `createDocumentAsynchronously` must return the document status, such as a status URL that could display the status of the document creation. All of the built-in document production types support asynchronous production, but asynchronous document production is not called from the default user interface or business rules.

Asynchronous document production increases the risk of concurrent data change exceptions (CDCE) because different threads might need to access the same data at the same time. The CDCE risk is greater for documents created by users in the web application, rather than by activities or Gosu rules. If some type of document production is extremely long or requires external servers, consider asynchronous production for that type of document to increase application responsiveness.

For documents created asynchronously, your `createDocumentAsynchronously` method must put the newly created contents into the DMS. Next, your external system can use web service APIs to add the document to notify ClaimCenter that the document now exists.

If your code to add the document is running on the ClaimCenter server, use methods on the entity to add the document, as shown in the following code statement.

```
newDocumentEntity = Claim.addDocument
```

In either case, immediately throw an exception if any part of your creation process fails.

Configuring document production implementation mapping

In the default configuration, document production configuration is defined by the registry for the `IDocumentProduction` plugin in the Plugins editor in Studio in the section known as plugin parameters. The list of parameters defines either a template type or a MIME type. That value maps to a corresponding `IDocumentProduction` plugin implementation.

A typical document production configuration includes parameters for template types that reference the built-in `IDocumentProduction` implementations for common file types. For example, the template type named `application/msword` specifies `gw.plugin.document.impl.ServerSideWordDocumentProduction` as the implementation class.

Alternatively, you can match a document production class based on a custom document production type value that corresponds to the specific template that was chosen in the application.

Refer to the Plugins Registry for the `IDocumentProduction` plugin to see the full mapping of document production implementations. Review the plugin parameter list for the `IDocumentProduction` plugin. Each parameter is a MIME type name, and its value is the fully-qualified name of the implementation class.

In the base configuration, ClaimCenter determines which `IDocumentProduction` implementation to use to produce a document from a specific template by following this procedure within its default `IDocumentProduction` plugin implementation.

1. ClaimCenter searches the plugin parameters for a document template type value in the plugin parameter list that matches the text in the `documentProductionType` property of the template. If a match is found, ClaimCenter proceeds with document production using the specified `IDocumentProduction` implementation class in that plugin parameter value.
2. If ClaimCenter does not find a match for the document production type, ClaimCenter searches the plugin parameters for a MIME type that matches the `MimeType` property of the template. If a match is found, ClaimCenter proceeds with document production using the specified `IDocumentProduction` implementation class in that plugin parameter value.
3. If no match is found, document production fails. If this happens, review the configuration of the `IDocumentProduction` plugin parameters and the MIME type configuration.

Add a custom MIME type for document production

About this task

To add a custom MIME type for document product, perform the following steps.

Procedure

1. In `config.xml`, add the new MIME type to the `<mimetypemapping>` section. The information for each `<mimetype>` element contains the following attributes.
 - `name` – The name of the MIME type. Use the same name as in the plugin registry, such as `text/plain`.
 - `extension` – The file extensions to use for the MIME type.
If more than one extension applies, use a pipe symbol (" | ") to separate them. ClaimCenter uses this information to map between MIME types and file extensions. To map from a MIME type to a file extension, ClaimCenter uses the first extension in the list. To map from file extension to MIME type, ClaimCenter uses the first `<mimetype>` entry that contains the extension.
 - `icon` – The image file for documents of this MIME type. At runtime, the image file must be in the `SERVER/webapps/cc/resources/images` directory.
2. Add the MIME type to the configuration of the application server, if required. Ensure that the MIME type is not in the list already before you try to add it. How you add a MIME type depends on the brand of application server. Refer to your application server documentation for details. For Tomcat, configure MIME types in the `web.xml` configuration file by using `<mime-mapping>` elements.

Licensing for server-side document production

This topic includes the following:

- “Licensing for Microsoft Excel and Word document production” on page 232
- “Licensing for PDF document production” on page 233

Licensing for Microsoft Excel and Word document production

For server-side document production of Microsoft Word and Microsoft Excel files, ClaimCenter includes license files in the following paths.

```
ClaimCenter/configuration/config/security/excel.lic
ClaimCenter/configuration/config/security/word.lic
```

Never remove or alter these license files. If changes are needed, replacements will be provided by Guidewire Customer Support.

Licensing for PDF document production

Before you begin

In the ClaimCenter base configuration, server-side PDF document production is implemented by software from the company Big Faceless Organization (BFO). PDF documents can be generated with or without a BFO license key. However, without a license key, the generated document will contain a large watermark that reads "DEMO" on each page. To remove the watermark, obtain a BFO license key through Guidewire Customer Support.

Note that a BFO license key applies only to PDF document production. When generating a PDF from other sources, such as when printing or exporting a ListView, the Apache Formatting Objects Processor (FOP) engine is used. The Apache FOP engine is separate from the BFO engine and does not require a BFO license key. Consequently, PDF files generated with the FOP engine never have a "DEMO" watermark.

About this task

To configure PDF production, perform the following steps.

Procedure

1. Receive an Authorization Form by contacting your Customer Support Partner to request a PDF production license for ClaimCenter.
2. Fill out the Authorization Form.
3. Return the filled-out form to Guidewire prior to issuing the license.
4. The Guidewire support engineer requests license keys from the appropriate departments.
5. Once the license keys are obtained, Guidewire emails the designated customer contact (per the information on the form) with the license information. If you have additional questions about this process, contact Guidewire Support>.
6. In a clustered environment, set the license key value in the `PDFMergeHandlerLicenseKey` parameter in `config.xml` for every server.
7. In the Project window, navigate to **Configuration > config > Plugins > registry**, and then open `IPDFMergeHandler.gwp`. Although you must edit the Plugins registry in Studio, `IPDFMergeHandler` does not correspond to a plugin interface that is intended for you to implement.
8. Ensure the implementation class is set to `gw.plugin.document.impl.BFOMergeHandler`.
9. In the list of plugin parameters set the plugin parameter `LicenseKey` value to your server key. The `LicenseKey` value is empty in the default configuration. You must acquire your own BFO licenses from customer support.

Write a document template descriptor and install a template

To support document production you must create the following types of files for every template.

- Document template descriptor – A template descriptor file describes metadata about the template and how to generate text to insert into the template. The document template descriptor describes a document template, such as a Microsoft Word mail merge template, an Adobe PDF form, or a Gosu template. There may be multiple of these, each of which represents a locale.
- Document template source file – A source file of the appropriate type, such as an RTF file for a Microsoft Word document. To support multiple language versions of the same document, there will be more than one of these template source files for a single descriptor file. The additional language versions of the file are in subdirectories by locale name.

A plugin interface called `IDocumentTemplateDescriptor` defines the API for an object that represents a document template descriptor. There is a built-in plugin implementation that loads document template descriptor data from an XML file from the local file system. For typical use, you just need to write one XML file for each document template descriptor.

The base configuration reads template descriptors from an XML file. In most cases, it is best to use this implementation of the `IDocumentTemplateDescriptor` interface, rather than modify the code. For typical requirements, you do not need to modify the code that reads or writes the XML file. Some information in this documentation topic is included for the rare case that you might need to write your own implementation of `IDocumentTemplateDescriptor`.

A template descriptor contains several categories of information.

- General template metadata – Template metadata is metadata about the template itself, not the file. Template metadata includes the template ID, and the template name. Some metadata helps filter the list of templates.

date range

Calendar dates that limit the availability of the template.

keywords

Searchable keywords for this template.

scope

Specifies whether the template is usable only in the user interface, or also is available for programmatic use. `UI` indicates only for use in the application user interface. `ALL` represents for all scopes.

business context metadata

Additional fields for filtering by business context, such as applicability only to a specific state.

- Required symbols – A list of required programmatic symbols that filter the template list to only the templates that make sense in this context in the application. For example, if the `Claim` symbol is required, only application contexts that provide a `Claim` symbol with a non-null value can use this document production template. Any Gosu expressions in the template must not rely on symbols that are not in this list. You can run template validation to confirm Gosu expressions in the template do not use symbols that are not required. You can validate templates using web services or equivalent command line tools.
- Document metadata defaults – Document metadata defaults are attributes that are applied to documents after their creation from the template, or as part of their creation, for example the default document status.
- Context objects and default values – Context objects are Gosu objects (including entity instances) that a template can use to generate form data. The template can use the object directly (such as `String` value) or might extract one or more properties as text form fields in a new document. For example, an email document template might include `To` and `CC` recipients as context objects of the type `Contact`. Every context object has a default value, which will be used if no alternative is chosen. The template descriptor provides a Gosu expression that gets a list of legal alternative values. In the user interface, the user can select alternatives, such as all contacts associated with a claim.
- Field names and values – Each descriptor defines a set of template field names and values to insert into the document template, including optional formatting information. Effectively, this describes which ClaimCenter data values to merge into which fields within the document template. For example, an email document template might have `To` and `CC` recipients as context objects called `To` and `CC` of type `Contact`. To extract the name of the insured person, the template might add a context object called `InsuredName` that extracts the value with the Gosu expression `To.DisplayName`. When producing documents from the user interface, the user can choose alternatives. For document production from Gosu, the default values are used.

The `IDocumentTemplateDescriptor` interface is closely tied to the XML file format, which corresponds to the base configuration implementation of the `IDocumentTemplateSerializer` interface. The `IDocumentTemplateDescriptor` API consists mostly of property getters, with one setter for `DateModified` and some additional utility methods.

Example simple template descriptor XML file with no context objects

```
<?xml version="1.0" encoding="UTF-8"?>
<DocumentTemplateDescriptor
    id="SamplePage.gosu.htm"
    name="Gosu Sample Web Page"
    description="The initial contact reservation rights letter/template."
    type="letter_sent"
    mime-type="text/html"
    keywords="CA, reservation">
```

```
<FormField name="ClaimNumber">Claim.ClaimNumber</FormField>
</DocumentTemplateDescriptor>
```

Example template descriptor XML file with context objects

The elements and attributes used in the example XML file are described in subsequent topics.

```
<?xml version="1.0" encoding="UTF-8"?>
<DocumentTemplateDescriptor
    id="SamplePage.gosu.htm"
    name="Gosu Sample Web Page"
    description="The initial contact reservation rights letter/template."
    type="letter_sent"
    lob="GLLine"
    state="CA"
    mime-type="text/html"
    keywords="CA, reservation">

    <ContextObject name="To" type="Contact">
        <DefaultObjectValue>Claim.maincontact</DefaultObjectValue>
        <PossibleObjectValues>Claim.getRelatedContacts()</PossibleObjectValues>
    </ContextObject>
    <ContextObject name="From" type="Contact">
        <DefaultObjectValue>Claim.AssignedUser.Contact</DefaultObjectValue>
        <PossibleObjectValues>Claim.getRelatedUserContacts()</PossibleObjectValues>
    </ContextObject>
    <ContextObject name="CC" type="Contact">
        <DefaultObjectValue>Claim.reporter</DefaultObjectValue>
        <PossibleObjectValues>Claim.getRelatedContacts()</PossibleObjectValues>
    </ContextObject>

    <FormFieldGroup name="main">
        <DisplayValues>
            <DateFormat>MMM dd, yyyy</DateFormat>
        </DisplayValues>
        <FormField name="ClaimNumber">Claim.ClaimNumber</FormField>
        <FormField name="InsuredName">To.DisplayName</FormField>
        <FormField name="InsuredAddress1">To.PrimaryAddress.AddressLine1</FormField>
        <FormField name="InsuredCity">To.PrimaryAddress.City</FormField>
        <FormField name="InsuredState">To.PrimaryAddress.State</FormField>
        <FormField name="InsuredZip">To.PrimaryAddress.PostalCode</FormField>
        <FormField name="CurrentDate">gw.api.util.DateUtil.currentDate()</FormField>
        <FormField name="ClaimNoticeDate">Claim.LossDate</FormField>
        <FormField name="AdjusterName">From.DisplayName</FormField>
        <FormField name="AdjusterPhoneNumber">From.WorkPhone</FormField>
        <FormField name="InsuranceCompanyName">Claim.Policy.UnderwritingCo</FormField>
        <FormField name="InsuranceCompanyAddress">From.PrimaryAddress.AddressLine1</FormField>
        <FormField name="InsuranceCompanyCity">From.PrimaryAddress.City</FormField>
        <FormField name="InsuranceCompanyState">From.PrimaryAddress.State</FormField>
        <FormField name="InsuranceCompanyZip">From.PrimaryAddress.PostalCode</FormField>
    </FormFieldGroup>
</DocumentTemplateDescriptor>
```

Template descriptor attributes

The following table lists attributes on the `<DocumentTemplateDescriptor>` element, which is the main element of the default XML format used by the built-in implementation of the `IDocumentTemplateDescriptor` plugin interface.

Unless indicated otherwise in the Description column, the attribute appears in the `IDocumentTemplateDescriptor` interface as a property getter and setter with the same name as the XML attribute.

Document template descriptor attribute as it appears in the descriptor XML file that you must create for new templates	Required?	Description
<code>id</code>	Required	The unique ID of the template. In the <code>IDocumentTemplateDescriptor</code> interface, this is the <code>templateId</code> property.
<code>name</code>	Required	A human-readable name for the template.

Document template descriptor attribute as it appears in the descriptor XML file that you must create for new templates	Required?	Description
identifier	Optional	An additional human-readable identifier for the template. This often corresponds to a well-known domain-specific document code. For example, to indicate a state-mandated form for this template.
scope	Optional	<p>The contexts that this template supports. The following values are supported.</p> <ul style="list-style-type: none"> • ui – The document template must only be used from the document creation user interface. • gosu – The document template must only be used from rules or other Gosu and must not appear in a list the user interface template. • all – The document template may be used from any context.
description	Required	A human-readable description of the template and/or the document it creates.
mime-type	Required	<p>The type of document to create from this document template. In the built-in implementation of document source, this determines which <code>IDocumentProduction</code> implementation to use to create documents from this template. Also see <code>documentProductionType</code>.</p> <p>In the <code>IDocumentTemplateDescriptor</code> interface, this is the <code> mimeType </code> property.</p>
production-type	Optional	<p>If present, a specified document production type indirectly specifies which implementation of <code>IDocumentProduction</code> to use to create a new document from the template. This is not the only way to select a document production plugin implementation. You can also use a MIME type, see the row for <code> mime-type </code>.</p> <p>Specify a unique production type <code>String</code> value. Next, in Studio, open the Plugins editor for the <code>IDocumentProduction</code> interface. Add a plugin parameter with the parameter name matching the value that you set for <code> production-type </code>. Finally, set the value of that plugin parameter to the fully-qualified name of the implementation of <code>IDocumentProduction</code> that handles this production type.</p> <p>In the <code>IDocumentTemplateDescriptor</code> interface, this is the <code> documentProductionType </code> property.</p>
date-modified	Optional	<p>The date the template was last modified. In the default implementation, this is set from the information on the XML file itself. Both getter and setter methods exist for this property so that the date can be set by the <code>IDocumentTemplateSource</code> implementation. This property is not present in the XML file. However, the built-in implementation of the <code>IDocumentTemplateDescriptor</code> interface generates this property.</p> <p>In the <code>IDocumentTemplateDescriptor</code> interface, this is the <code> dateModified </code> property.</p>
date-effective date-expires	Required	<p>The effective and expiration dates for the template. If you search for a template, ClaimCenter displays only those for which the specified date falls between the effective and expiration dates. However, this does not support different versions of templates with the same ID. The ID for each template must be unique. You can still create documents from templates from Gosu (from PCF files or rules) independent of these date values. Gosu-based document creation uses template IDs but ignores effective dates and expiration dates.</p> <p>In the <code>IDocumentTemplateDescriptor</code> interface, these are the <code> dateEffective </code> and <code> dateExpiration </code> properties.</p>
keywords	Required	A set of keywords to search for within the template. Delimit keywords with the comma character. Do not add space characters between keywords.
required-permission	Optional	The code of the <code>SystemPermissionType</code> value required for the user to see and use this template. Templates for which the user does not have the appropriate permission do not appear in the user interface. This setting does not prevent creation of a document by Gosu (PCF files or rules).

Document template descriptor attribute as it appears in the descriptor XML file that you must create for new templates	Required?	Description
		In the <code>IDocumentTemplateDescriptor</code> interface, this is the <code>requiredPermission</code> property.
mail-merge-type	Optional	<p>Optional configuration of pagination of Microsoft Word production. By default, ClaimCenter uses Microsoft Word catalog pagination, which correctly trims the extra blank page at the end. However, catalog pagination forbids template substitution in headers and footers. In contrast, standard pagination adds a blank page to the end of the file but enables template substitution in headers and footers. Set this attribute to the value <code>catalog</code> to use catalog pagination. To use standard pagination, do not set this attribute.</p> <p>In the <code>IDocumentTemplateDescriptor</code> interface, this is the <code>mailmergetype</code> property.</p>
type	Required	<p>Corresponds to the <code>DocumentType</code> typelist. Documents created from this template have their type fields set to this value.</p> <p>In the <code>IDocumentTemplateDescriptor</code> interface, this is the <code>templateType</code> property.</p>
default-security-type	Optional	<p>Security type in the <code>DocumentSecurityType</code> typelist. This is the security type that becomes the default value for the corresponding document metadata fields for documents created using this template. Use this in conjunction with the information in <code>security-config.xml</code>.</p> <p>In the <code>IDocumentTemplateDescriptor</code> interface, this is the <code>defaultSecurityType</code> property.</p>
lob	Required	Line of business name.
section	Optional	The section to which this document belongs, if any.
state	Required	State, such as CA for California.
required-symbols	Required	<p>A list of programmatic symbols that are required in this context for this template to work correctly at run time. For the user interface, this is a filtering mechanism. For example, if the <code>Claim</code> symbol is required, only application contexts that provide a <code>Claim</code> symbol with a non-null value can use this document production template. Gosu expressions in the template must not rely on symbols that are not in this list. Also see the row for <code>availableSymbols</code>.</p> <p>At run time, the list of symbols of this context is provided by the application. If you trigger document creation in the user interface, the PCF page defines the symbols. If you trigger document creation from Gosu code such as from rule sets, the symbols are provided as a <code>java.util.Map</code>.</p> <p>In the <code>IDocumentTemplateDescriptor</code> interface, this is the <code>requiredSymbols</code> property.</p>
availablesymbols	Required	<p>A list of programmatic symbols that are possible in this user interface or programmatic context. Any Gosu expressions in the template must not rely on symbols that are not in this list. You can run template validation to confirm Gosu expressions in the template do not use symbols that are not available. You can validate templates using web services or equivalent command line tools.</p> <p>In the <code>IDocumentTemplateDescriptor</code> interface, this is the <code>availableSymbols</code> property.</p>

Form fields and field groups

To document templates that include Gosu expressions, you need to reference business data such as the current `Claim` entity. To insert text into a form field in a template, you define your list of form fields within the template descriptor. Each form field in the template descriptor defines a Gosu expression that generates the text for that field.

In the descriptor file, form fields create a mapping between ClaimCenter data and fields in the document template. For example, to merge the claim number into the form field `ClaimInfo`, use the following expression.

```
<FormField name="ClaimInfo">Claim.ClaimNumber</FormField>
```

A form field can reference business data as objects referenced by symbols, which are programmatic labels for objects such as entity instances. At run time, the list of symbols of this context is provided by the application. If you trigger document creation in the user interface, the current PCF page defines the symbols. If you trigger document creation from Gosu code such as from rule sets, the symbols and values are provided as a `java.util.Map`. For example, form fields can reference a relevant claim by using the claim symbol. For example, get properties or call methods with code such as `claim.MyProperty` or `claim.myMethod()`.

Similarly, ClaimCenter, if a related exposure or claimant is selected in the document creation screen, that object also is available through the symbol `RelatedTo`. For example, `RelatedTo.DisplayName`.

`<FormField>` elements can contain any valid Gosu expression. If the logic is complex, you can call Gosu enhancements or other APIs that encapsulate the logic. For example, you can create an Gosu enhancement method `Claim.getClaimInfoSummary`. Reference that function in a form field as follows.

```
<FormField name="ClaimInfoSummary">Claim.getClaimInfoSummary()</FormField>
```

Typically, the Gosu in the `<FormField>` elements refer to a context object defined earlier in the file.

To add a prefix before the generated output from the Gosu expression, add the additional attribute `prefix` and set to a simple text value. To add a suffix, use the `suffix` attribute. For example, to add the prefix “The claim information is” and a period character as the suffix, specify the field as shown below.

```
<FormField name="MainContactName"
  prefix="The claim information is "
  suffix=".">
  Claim.getClaimInformation()
</FormField>
```

You must encapsulate your `<FormField>` elements in sets called field groups, even if there is only one field in the group.

Field groups

In the XML file for a template descriptor, you must encapsulate your `<FormField>` elements in sets called field groups, even if there is only one field in the group.

Form field groups are implemented as a `<FormFieldGroup>` element that contain the following elements:

- optional `<DisplayValues>` elements that customize the display of dates and other special values within this field group.
- one or more `<FormField>` elements that represent form fields.

A descriptor file can have any number of `<FormFieldGroup>` elements. Define multiple field groups to define common display attributes across only a specific subset of form fields. For example, perhaps only some fields share the same a date string format or special handling of `null`, `true`, or `false`.

Display values

Define one or more `<DisplayValues>` elements within a `<FormFieldGroup>` element to customize display of date values and other special values like the values `null`, `true`, or `false`. For example, you could use a `<DisplayValues>` definition for a group to find the value `null` and instead output the text “No coverage”. The following table lists the elements you can insert inside a `<DisplayValues>` element. You can define more than one of these child elements inside the `<DisplayValues>` element.

Child element of <DisplayValues>	Description
<NullDisplayValue>	Customizes the display of null values. The content of the element is the value to substitute.
	Note that for typical Gosu entity path expressions using the period character, if any object in the path evaluates to null, the expression silently evaluates to null. This feature is called null safety. For example, if an entity instance Obj has the value null, then Obj.SubObject.
	Subsubobject evaluates to null. Use the <NullDisplayValue> to display something better if any part of the a field path expression is null.
	ClaimCenter also uses the NullDisplayValue if an invalid array index is encountered. For example, the expression "MyEntity.doctor[0].DisplayName" results in displaying the NullDisplayValue if the doctor array is empty.
	The null safety does not work with typical method calls on a null expression. For example, the expression Obj.SubObject.Field1() throws an exception if Obj is null.
<TrueDisplayValue>	Customizes the display of true values. The content of the element is the value to substitute.
<FalseDisplayValue>	Customizes the display of false values. The content of the element is the value to substitute.
<NumberFormat>	Customizes the display of numeric values.
<DateFormat>	Customizes the display of date values. If you provide a <DateFormat> child of <DisplayValues>, do not provide a <TimeFormat>.
<TimeFormat>	Customizes the display of time values. If you provide a <TimeFormat> child of <DisplayValues>, do not provide a <DateFormat>.

Date value syntax

You can specify date values in the document template descriptor XML file in customizable formats.

Specify your required date syntax by using the <DateFormat> child element of <DisplayValues> in a <FormFieldGroup> element in the descriptor file. The following lines show an example.

```
<FormFieldGroup name="default">
  <DisplayValues>
    <DateFormat>MMM dd, yyyy</DateFormat>
  </DisplayValues>
  <FormField name="CurrentDate">gw.api.util.DateUtil.currentDate()</FormField>
</FormFieldGroup>
```

The following codes are used to specify date formats.

- a = AM or PM
- d = day
- E = Day in week (abbrev.)
- h = hour (24 hour clock)
- m = minute
- M = month (MMMM is the entire month name)
- s = second
- S = fraction of a second
- T = parse as time (ISO8601)
- y = year
- z = Time Zone offset.

The following formats are available for systems in the English locale.

Date format	Example
MMM d, yyyy	Jun 3, 2018
MMMM d, yyyy	June 3, 2018
Note: Four M characters specify the entire month name.	
MM/dd/yy	10/30/19
MM/dd/yyyy	10/30/2019
MM/dd/yy hh:mm a	10/30/19 10:20 pm
yyyy-MM-dd HH:mm:ss.SSS	2018-06-09 15:25:56.845
yyyy-MM-dd HH:mm:ss	2018-06-09 15:25:56
yyyy-MM-dd'T'HH:mm:ss zzz	2018-06-09T15:25:56 -0700
EEE MMM dd HH:mm:ss zzz yyyy	Thu Jun 09 15:24:40 -0700 2018

The first three formats are the most commonly used because templates typically expire at the end of a particular day rather than at a particular time.

For text elements, such as month names, ClaimCenter requires the text representations of the values to match the current international locale settings of the server. For example, if the server is in the French locale, you must provide the month April as "Avr", which is short for Avril, the French word for April.

When processing a document template with the `template_tools` command-line tool, the dates in the input files are preserved, but the date format might change.

Context objects

In addition to the symbols provided by the API caller or the context of the user interface, a template descriptor can define and reference a custom object called a "context object." A context object creates a new shorthand symbol that can be referenced by `FormField` expressions. A template descriptor can define multiple context objects.

Without context objects, you would have access to only one or two high-level root objects, which would get challenging. For example, suppose you want to address a claim acknowledgment letter to the main contact on the claim. Without context objects, each `FormField` element would have to repeat a prefix many times.

```
<FormField name="ToName">Claim.MainContact.DisplayName</FormField>
<FormField name="ToCity">Claim.MainContact.PrimaryAddress.City</FormField>
<FormField name="ToState">Claim.MainContact.PrimaryAddress.State</FormField>
```

This code is difficult to read because of its lengthy prefixes. Instead, you can simplify template code by defining a `ContextObject` element that refers to the intended recipient. Each `ContextObject` must have the following required attributes.

- `name` - Unique ID specified as a `String`, such as "To"
- `type` - Object type specified as a `String`, such as "Contact"

The `ContextObject` element can include the following optional attributes.

- `display-name` - Text to show in the user interface. Default value is the `String` assigned to the `name` attribute.
- `allow-null` - Boolean value indicating whether an object value is required. If `false` then the object is a required field that must have a value assigned to it. Default is `true`.

Each `ContextObject` element must define a `DefaultObjectValue` element which specifies the element's default value. The value can be a Gosu expression that returns the type specified in the `ContextObject` element's `type` attribute. The `DefaultObjectValue` can reference another `ContextObject` by specifying the `ContextObject` element's `name`. A `ContextObject` must be defined before it can be referenced.

A sample ContextObject is defined below.

```
<ContextObject name="To" type="Contact">
  <DefaultObjectValue>Claim.MainContact</DefaultObjectValue>
</ContextObject>
```

The name attribute value of "To" defined in the sample ContextObject can be used to simplify the original FormField elements.

```
<FormField name="ToName">To.DisplayName</FormField>
<FormField name="ToCity">To.PrimaryAddress.City</FormField>
<FormField name="ToState">To.PrimaryAddress.State</FormField>
```

The ContextObject also defines a list of possible values in a PossibleObjectValues element. The possible values are specified as a Gosu expression that evaluates to an array. Typically the array contains entity instances, but that is not a requirement.

Be aware that ClaimCenter does not verify and enforce that the type of the PossibleObjectValues matches the type used by the DefaultObjectValue. While this makes it possible to have two different types for a single ContextObject, Guidewire strongly recommends against this approach. If the specified types do not match, the relevant FormField expressions must be written so they work correctly at run time with multiple types for the ContextObject.

A sample ContextObject definition that specifies default and possible values is shown below.

```
<ContextObject name="To" type="Contact">
  <DefaultObjectValue>Claim.MainContact</DefaultObjectValue>
  <PossibleObjectValues>Claim.getRelatedContacts()</PossibleObjectValues>
</ContextObject>
```

Context objects have an additional purpose. In the user interface of the application, you can manually specify the value of each context object within the user interface among multiple relevant possibilities. For example, the likely recipient of an email can be set as a default, but the template can offer all contacts on the claim as alternatives.

If you select a document template from the chooser, ClaimCenter displays a series of choices, one for each context object. The name of the context object appears as a label on the left, and the default value appears on the right. The template defines the default value to use, but the user can override it by choosing from the list of possible object values.

Context object type reference

Context objects must be of one of the following types in its type attribute.

Context object type	Meaning
EntityName	The name of a ClaimCenter entity type such as Claim or Activity. If that type of entity has a special type of picker, the application displays the special picker. For example, if you set the context object type to "Contact", users can use the Contact picker to search for a different value. Similarly, if you set the context object type to "User", ClaimCenter displays a user picker.
Bean	To indicate support for any keyable entity, provide the literal text value "Bean". This is useful for heterogeneous lists of objects.
TypeListName	The name of a ClaimCenter typelist, such as "YesNo".
string	This value specifies a text value that appears in the user interface as a single line of text. The <DefaultObjectValue> tag must indicate the default text for this context object. This context object type always ignores the <PossibleObjectValues> tag. To use this context object type, the type value string must be all lower case.
text	Appears in the user interface as several lines of text. The <DefaultObjectValue> tag must be present, and its contents indicates the default text for this context object. If you use this, ClaimCenter ignores the <PossibleObjectValues> tag.

Context object type Meaning	
To use this context object type, the type value <code>text</code> must be all lower case.	
date	A date of type <code>java.util.Date</code> .
To use this context object type, the type value <code>date</code> must be all lower case.	

The `type` attribute on the `ContextObject` is used to indicate how the user interface presents the object in the document creation user interface. Valid options include: `String`, `text`, `Contact`, `User`, `Entity`, `Claim`, or any other ClaimCenter entity name or typekey type.

If the context object specifies type `string`, then the user would typically be given a single-line text entry box.

If the context object specifies type `text`, the user sees a larger text area. However, if the `ContextObject` definition includes a `PossibleObjectValues` tag containing Gosu that returns a `Collection` or array of `String` objects, the user interface displays a selection picker. For example, use this approach to offer a list of postal codes from which to choose. If the object is of type `Contact` or `User`, in addition to the drop-down box, you see a picker button to search for a particular contact or user. All other types (`Entity` is the default if none is specified) are presented as a drop-down list of options. If the `ContextObject` is a typekey type, then the default value and possible values fields must generate Gosu objects that resolve to `TypeKey` objects, not text versions of typecodes values.

There are a few instances in ClaimCenter system in which entity types and typekey types have the same name, such as `Contact`. In this case, Gosu assumes you mean the entity type. If you want the typelist type, or want clearer code, use the fully qualified name of the form `entity.EntityName` or `typekey.TypeKeyName`.

Gosu APIs on template descriptor instances

For typical requirements, you do not need to manipulate document template descriptors from Gosu code. If you do, you can use API enhancement methods on the `IDocumentTemplateDescriptor` interface.

Use the following properties and methods to access context objects more easily from Gosu.

- `ContextObjectNames` – Returns an array of `String` values that are the set of context object names defined in the document template.
- `getContextObjectType(String objName)` – Returns the type of the specified context object. Possible values include "string", "text", "Entity" (for any entity type), or the name of an entity type such as "Claim".
- `boolean getContextObjectAllowsNull(String objName)` – Returns `true` if `null` is a legal value, `false` otherwise.
- `String getContextObjectDisplayName(String objName)` – Returns a human-readable name for the given context object, to display in the document creation user interface.
- `String getContextObjectDefaultValueExpression(String objName)` – Returns a Gosu expression which evaluates to the desired default value for the context object. Use this to set the default for the document creation user interface, or as the value if a document is created automatically.
- `String getContextObjectPossibleValuesExpression(String objName)` – Returns a Gosu expression that evaluates to the desired set of legal values for the given context object. Used to display a list of options for the user in the document creation user interface.
- `getCompiledContextObjectDefaultValueExpression` – Returns a compiled Gosu expression for a context object default value. In the rare case you need to re-implement the `IDocumentTemplateDescriptor` interface, these two methods require special return result types. For information about implementing these methods, contact Guidewire Customer Support.
- `getCompiledContextObjectPossibleValuesExpression` – Returns a compiled Gosu expression for a context object list of possible values. See the note about return type of the method `getCompiledContextObjectDefaultValueExpression`.
- `getFormFieldCompiledExpression` – Returns a compiled Gosu expression for a form field. See the note about return type of the method `getCompiledContextObjectDefaultValueExpression`.

- **MetadataPropertyNames** - This property returns the set of extra metadata properties that exist in the document template definition. This method is used in conjunction with `getMetadataPropertyValue` as a flexible extension mechanism. You can add arbitrary new fields to document template descriptors. ClaimCenter passes new properties to the internal entities that display document templates in the user interface. Also, if the extra property names correspond to properties on the Document entity, the ClaimCenter passes values to documents created from the template. In the XML file, this is represented by additional attributes on the `<DocumentTemplateDescriptor>` element.
- `getMetadataPropertyValue(String)` - This method gets a property value. The method takes one argument, which is a property name as a `String` value. See the `MetadataPropertyNames` property. In the XML file, this is represented by additional attributes on the `<DocumentTemplateDescriptor>` element.

See also

- “Add custom attributes document template descriptor XML format” on page 244

Template descriptor fields related to form fields and values to merge

Form fields dictate a mapping between ClaimCenter data and the merge fields in the document template.

For example, you might want to merge the claim number into a document field using the simple Gosu expression `"Claim.ClaimNumber"`.

The full set of template descriptor fields relating to form fields are as follows.

- `String[] getFormFieldNames()` – Returns the set of form fields defined in the document template. Refer to the following XML document format for more information on what the underlying configuration looks like.
- `String getFormFieldValueExpression(String fieldName)` – Returns a Gosu expression that evaluates to the desired value for the form field. The Gosu expression is usually written in terms of one or more Context Objects, but any legal Gosu expression is allowed.
- `String getFormFieldDisplayValue(String fieldName, Object value)` – Returns the string to insert into the completed document given the field name and the value. The value is typically the result of evaluating the expression returned from the method `getFormFieldValueExpression`. Use this method to rewrite values if necessary, such as substituting text. For example, display text that means “not applicable” (`"<n/a>"`) instead of null, or format date fields in a specific way.

XML format for document template descriptors

The implementation of `IDocumentTemplateSerializer` in the base configuration intentionally uses an XML format that closely matches the fields in the `DocumentTemplateDescriptor` interface. The purpose of `IDocumentTemplateSerializer` is to serialize template descriptors and so that you can define the templates within simple XML files. The XML format is suitable in typical implementations. ClaimCenter optionally supports different implementations that might directly interact with a document management system storing the template configuration information. However, the base configuration XML format is sufficient for most requirements.

The XML format described in this section is basically a serialization of the fields in the `IDocumentTemplateDescriptor` interface. In the base configuration, the `IDocumentTemplateDescriptor` and `IDocumentTemplateSerializer` classes implement the serialization.

The base configuration `IDocumentTemplateSerializer` is configured by a file named `document-template.xsd`. The base configuration uses XML similar to the following example.

```
<?xml version="1.0" encoding="UTF-8"?>
<DocumentTemplateDescriptor
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="document-template.xsd"
    id="ReservationRights.doc"
    name="Reservation Rights"
    description="The initial contact letter/template."
    type="letter_sent"
    lob="GL"
    state="CA"
    mime-type="application/msword"
```

```

date-effective="Apr 3, 2019"
date-expires="Apr 3, 2020"
required-symbols="Claim"
keywords="CA, reservation">
<ContextObject name="To" type="Contact">
  <DefaultObjectValue>Claim.MainContact</DefaultObjectValue>
  <PossibleObjectValues>Claim.getRelatedContacts()</PossibleObjectValues>
</ContextObject>
<ContextObject name="From" type="Contact">
  <DefaultObjectValue>Claim.AssignedUser.Contact</DefaultObjectValue>
  <PossibleObjectValues>Claim.getRelatedUserContacts()</PossibleObjectValues>
</ContextObject>
<ContextObject name="CC" type="Contact">
  <DefaultObjectValue>Claim.Driver</DefaultObjectValue>
  <PossibleObjectValues>Claim.getRelatedContacts()</PossibleObjectValues>
</ContextObject>
<FormFieldGroup name="main">
  <DisplayValues>
    <NullDisplayValue>No Contact Found</NullDisplayValue>
    <TrueDisplayValue>Yes</TrueDisplayValue>
    <FalseDisplayValue>No</FalseDisplayValue>
  </DisplayValues>
  <FormField name="ClaimNumber">Claim.ClaimNumber</FormField>
  <FormField name="InsuredName">To.DisplayName</FormField>
  <FormField name="InsuredAddress1">To.PrimaryAddress.AddressLine1</FormField>
  <FormField name="InsuredCity">To.PrimaryAddress.City</FormField>
  <FormField name="InsuredState">To.PrimaryAddress.State</FormField>
  <FormField name="InsuredZip">To.PrimaryAddress.PostalCode</FormField>
  <FormField name="CurrentDate">Libraries.Date.currentDate()</FormField>
  <FormField name="ClaimNoticeDate">Claim.LossDate</FormField>
  <FormField name="AdjusterName">From.DisplayName</FormField>
  <FormField name="AdjusterPhoneNumber">From.WorkPhone</FormField>
  <FormField name="InsuranceCompanyName">Claim.Policy.UnderwritingCo</FormField>
  <FormField name="InsuranceCompanyAddress">From.PrimaryAddress.AddressLine1</FormField>
  <FormField name="InsuranceCompanyCity">From.PrimaryAddress.City</FormField>
  <FormField name="InsuranceCompanyState">From.PrimaryAddress.State</FormField>
  <FormField name="InsuranceCompanyZip">From.PrimaryAddress.PostalCode</FormField>
</FormFieldGroup>
</DocumentTemplateDescriptor>

```

At run time, this XSD is referenced from a path relative to the module config/resources/doctemplates directory. To change this value, in the plugin registry for this plugin interface, in Guidewire Studio in the Plugins editor, set the DocumentTemplateDescriptorXSDLLocation parameter. To use the default XSD in the default location, set that parameter to the value "document-template.xsd".

The attributes on the DocumentTemplateDescriptor element correspond to the properties on the IDocumentTemplateDescriptor API.

Add custom attributes document template descriptor XML format

If you use the default XML template descriptor that is implemented by built-in classes, you can add custom attributes to the XML file. To extend the set of attributes on document templates, a few steps are required.

First, modify the document-template.xsd file, or create a new one. The location of the .xsd file used to validate the document is specified by the DocumentTemplateDescriptorXSDLLocation parameter within the application config.xml file. This location is specified relative to the WEB_APPLICATION/WEB-INF/platform directory in the deployed web application directory.

Any number of attributes can be added to the definition of the DocumentTemplateDescriptor element. This is the only element which can be modified in this file, and the only supported way in which it can be modified.

For example, you could add an attribute named **myattribute** as shown in bold in the following example.

```

<xsd:element name="DocumentTemplateDescriptor">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref="ContextObject" minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element ref="HtmlTable" minOccurs="0" maxOccurs="unbounded" />
      <xsd:element ref="FormFieldGroup" minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:string" use="required"/>
    <xsd:attribute name="name" type="xsd:string" use="required"/>
    <xsd:attribute name="identifier" type="xsd:string" use="optional"/>
    <xsd:attribute name="scope" use="optional">
      <xsd:simpleType>
        <xsd:restriction base="xsd:string">
          <xsd:enumeration value="all"/>
          <xsd:enumeration value="gosu"/>
          <xsd:enumeration value="ui"/>
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:attribute>
  </xsd:complexType>
</xsd:element>

```

```

</xsd:restriction>
</xsd:simpleType>
</xsd:attribute>
<xsd:attribute name="password" type="xsd:string" use="optional"/>
<xsd:attribute name="description" type="xsd:string" use="required"/>
<xsd:attribute name="type" type="xsd:string" use="required"/>
<xsd:attribute name="lob" type="xsd:string" use="required"/>
<xsd:attribute name="myattribute" type="xsd:string" use="optional"/>
<xsd:attribute name="section" type="xsd:string" use="optional"/>
<xsd:attribute name="state" type="xsd:string" use="required"/>
<xsd:attribute name="mime-type" type="xsd:string" use="required"/>
<xsd:attribute name="date-modified" type="xsd:string" use="optional"/>
<xsd:attribute name="date-effective" type="xsd:string" use="required"/>
<xsd:attribute name="date-expires" type="xsd:string" use="required"/>
<xsd:attribute name="keywords" type="xsd:string" use="required"/>
<xsd:attribute name="required-permission" type="xsd:string" use="optional"/>
<xsd:attribute name="default-security-type" type="xsd:string" use="optional"/>
</xsd:complexType>
</xsd:element>

```

Next, enable the user to search on the **myattribute** attribute and see the results. Add an item with the same name (**myattribute**) to the PCF files for the document template search criteria and results.

Now the new **myattribute** property shows up in the template search dialog. The search criteria processing happens in the **IDocumentTemplateSource** implementation. The default implementation, **LocalDocumentTemplateSource**, automatically handles new attributes by attempting an exact match of the attribute value from the search criteria. If the specified value in the descriptor XML file contains commas, it splits the value on the commas and tries to match any of the resulting values.

For example, if the value in the XML is **test**, then only a search for "test" or a search that not specifying a value for that attribute finds the template. If the value in the XML file is "**test,hello,purple**", then a search for any of "test", "hello", or "purple" finds that template.

As ClaimCenter creates the merged document, the application tries to match attributes on the document template with properties on the Document entity. For each matched property, ClaimCenter copies the value of the attribute in the template descriptor to the newly created Document entity. The user can either accept the default or change it to review the newly-created document.

Create your actual template file

Refer to the following table for inserting template form fields for each template type.

Template type	Inserting form fields
HTML, Gosu, RTF, or any other text-based format	<p>Insert one form field using text of the following format:</p> <p><%=FORM_FIELD_NAME%></p> <p>For example, for a form field called PolicyNumber:</p> <p><%=PolicyNumber%></p>
Microsoft Word	<p>To use a Microsoft Word file for document production, you must set up the Word file to use form fields. For example, suppose your template descriptor generates text for form fields called InsuredName and PolicyNumber. Your Word file must contain form fields called InsuredName and PolicyNumber.</p> <p>To set up form fields correctly, you need to create a temporary data source file that mimics the data that you are going to import from ClaimCenter. The data source file could be a simple comma-separated values (CSV) file that you generate in Microsoft Excel, and save as CSV format.</p> <p>Make the first line of the CSV file contain column headings that match your form fields. Continuing the earlier example, the CSV file must have two columns with the first lines identifying columns called InsuredName and PolicyNumber. To avoid Word errors caused by empty data source files, add one row of fake values for each form field, as shown in the following example.</p> <p style="padding-left: 40px;">InsuredName,PolicyNumber FakeName,FakePolicy</p> <ol style="list-style-type: none"> Start the Mail Merge wizard or chose Step by Step Merge Wizard. When the wizard user interface asks about using an existing list, click Browse....

Template type	Inserting form fields
	<ol style="list-style-type: none"> 3. Select your CSV file. 4. If you see a dialog that asks for the delimiter, choose comma (,). 5. In any place in the document that you want to add a form field, click the Insert Merge Field button in the Microsoft Word Ribbon user interface. From the dialog that appears, select the desired form field. For our previous example, select InsuredName or PolicyNumber.
Microsoft Excel	<p>For Microsoft Excel files, form fields are implemented using what Excel calls named ranges.</p> <ol style="list-style-type: none"> 1. Select a cell in the file. 2. Right-click on it. 3. Choose Name. 4. Type the name of the form field as defined in your template descriptor file.

Adding document templates and template descriptors to a configuration

In the base configuration, ClaimCenter keeps a set of document templates on the ClaimCenter application server. The document templates are initially in the following directory:

```
ClaimCenter/modules/configuration/config/resources/doctemplates
```

At run time, the document templates are typically in the following directory.

```
SERVER/webapps/cc/modules/configuration/config/resources/doctemplates
```

For each template, there are the following two files.

File	Naming	Example	Purpose
The template	Template regular file name	Test.doc	Contains the text of the letter, plus named fields that fill in with data from ClaimCenter.
The template descriptor	Template name with the suffix .descriptor	Test.descriptor	<p>This XML-formatted file provides various information about the template.</p> <ul style="list-style-type: none"> • How to search and find this template • Form fields that define what information populate each merge field • Form fields that define context objects • Form fields that define root objects to merge

To set up a new form or letter, you must create a template file and a template descriptor file. Then, deploy them to the **templates** directory on the web server.

To support localized templates, add extra descriptor files in subdirectories by locale code. For example, to add an additional Japanese document template of the earlier example, add the descriptor to the following directory.

```
ClaimCenter/modules/configuration/config/resources/doctemplates/jp/Test.doc.descriptor
```

See also

- An external system can retrieve and validate document templates. For information, see “Template web service” on page 144.

Document template descriptor optional cache

By default, ClaimCenter calculates the list of document templates from files locally on disk each time the application needs them. If you have only a small list of document templates, this is a quick process. However, if you have a large number of document templates, you can tell ClaimCenter to cache the list for better performance.

You might prefer to use the default behavior (no caching) during development, particularly if you are frequently changing templates while the application is running. However, for production, set the optional parameter in the document template source plugin to cache the list of templates.

To enable document template descriptor caching, perform the following steps.

1. In **Project** window in Guidewire Studio, navigate to **Configuration > config > Plugins > registry**, and then open `IDocumentTemplateSource.gwp`.
2. Under the plugin parameters editor in the right pane, add the `cacheDescriptors` parameter with the value `true`.

Creating new documents from Gosu rules

ClaimCenter can generate documents without user intervention from Gosu rules or from any other Gosu code.

The automatic form generation process is very similar to the manual document generation process, but effectively skips some steps previously described for manual form generation. Because there is no user interaction, choosing a template and parameter information happens in Gosu code, not the user interface.

To create a document, first create a map of values that specifies the value for each symbol. Using `java.util.HashMap` is recommended, but any `Map` type is legal. This value map must be non-null. The values in this map are unconstrained by either the default object value or the possible object values. Be careful to pass valid objects of the correct type.

If the default value for the context object (as defined by the template descriptor) is inappropriate, you can change it. Add a symbol to the map for that context object, for example the context object symbol called `Claim`. Set the value to the desired object, such as the appropriate `Claim` entity to use instead of the default.

Within Gosu rules, the Gosu class that handles document production is `gw.document.DocumentProduction`. It has methods for synchronous or asynchronous creation, which call the respective synchronous or asynchronous methods of the appropriate document production plugin, `createDocumentSynchronously` or `createDocumentAsynchronously`. Additionally there is a method to store the document: `createAndStoreDocumentSynchronously`. You can modify this Gosu class as needed.

If synchronous document creation fails, the `DocumentProduction` class throws an exception, which can be caught and handled appropriately by the caller. If document storage errors happen later, such as for asynchronous document storage, the document content storage plugin must handle errors appropriately. For example, the plugin could send administrative emails or create new activities using SOAP APIs to investigate the issues. Or you could design a system to track document creation errors and document management problems in a separate user interface for administrators. In the latter case, the plugin could register any document creation errors with that system.

If the synchronous document creation succeeds, next your code must attach the document to the claim by setting `Document.Claim`.

New documents always use the latest in-memory versions of entity instances at the time the rules run, not the versions as persisted in the database.

Be careful creating documents within Pre-Update Gosu rules or in other cases where changes can be rolled back due to errors (Gosu exceptions) or validation problems. If errors occur that roll back the database transaction even though rules added a document an external document management system, the externally-stored document is orphaned. The document exists in the external system but no ClaimCenter persisted data links to it.

Example document creation after sending an email

You can use code like the following to send a standard email and then create a corresponding document.

```
uses gw.document.DocumentProduction
uses gw.plugin.Plugins
```

```
// First, construct the email
var toContact : Contact = myClaim.Insured
var fromContact : Contact = myClaim.AssignedUser.Contact.Person
var subject : String = "Email Subject"
var body : String = "Email Body"

// Next, actually *send* the email
gw.api.email.EmailUtil.sendEmailWithBody(myClaim, toContact, fromContact, subject, body)

// Next, create the document that records the email
var document : Document = new Document(myClaim)
document.Claim = myClaim
document.Name = "Create by a Rule"
document.Type = "letter_sent"
document.Status = "draft"
// ...perhaps add more property settings here

// Create some "context objects"
var parameters = new java.util.HashMap()
parameters.put("To", toContact)
parameters.put("From", fromContact)
parameters.put("Subject", subject)
parameters.put("Body", body)
parameters.put("RelatedTo", myClaim)
parameters.put("Claim", myClaim)

// Create and store the document using the context objects
DocumentProduction.createAndStoreDocumentSynchronously("EmailSent.gosu.htm", parameters, document)
```

Important notes about cached document UIDs

If a new document is created and an error occurs within the same database transaction, the error typically causes the database transaction to roll back. This means that no database data was changed. However, if the local ClaimCenter transaction rolls back, there is no stored reference in the ClaimCenter database to the document unique ID (UID). The UID describes the location of the document in the external system. This information is stored in the Document entity in ClaimCenter in the same transaction, so the Document entity was not committed to the database. The new document in the external system is orphaned, and additional attempts to change ClaimCenter data regenerates a new, duplicate version of the document.

For the common case of validation errors, ClaimCenter avoids this problem. If a validatable entity fails validation, ClaimCenter saves the document UID in local memory. If the user fixes the validation error in that user session, ClaimCenter adds the document information as expected so no externally-stored documents are orphaned.

However, if other errors occur that cause the transaction to roll back (such as uncaught Gosu exceptions), externally-stored documents associated with the current transaction could be orphaned. The document is stored externally but the ClaimCenter database contains no Document entity that references the document UID for it. Avoiding orphaned documents is a good reason to ensure your Gosu rules properly catches exceptions and handles errors elegantly and thoroughly. Write good error-handling code and logging code always, but particularly carefully in document production code.

Geographic data integration

Guidewire ClaimCenter and Guidewire ContactManager provide an integration API for assigning a latitude and a longitude to an address. These two decimal numbers identify a specific location in degrees north or south of the equator and east or west of the prime meridian. The process of assigning these geographic coordinates to an address is called geocoding. Additionally, ClaimCenter and ContactManager support routing services, such as getting a map of an address and getting driving directions between two addresses, provided that the addresses are geocoded already.

Geocoding plugin integration

Guidewire ClaimCenter and Guidewire ContactManager use the Guidewire geocoding plugin to provide geocoding services in a uniform way, regardless of the external geocoding service that you use. The application requests geocoding services from the registered `GeocodePlugin` implementation. `GeocodePlugin` implementations typically do not apply geocode coordinates directly to addresses in the application database. Instead, the application requests geocode coordinates from the plugin, and the application determines how to apply them.

In addition, the `GeocodePlugin` interface supports routing services, such as retrieving driving directions and maps from external geocoding services that support these features. The interface also defines a method for reverse geocoding, which gets an address from geocode coordinates. The plugin interface defines methods that enable callers of the plugin to determine if the registered implementation supports routing services and reverse geocoding.

How ClaimCenter uses geocode data

ClaimCenter uses geocode data for actions in rules and for geographic searches. The base configuration of ClaimCenter supports the following high-level geocoding features:

- ClaimCenter user assignment and searching – ClaimCenter can assign claims to users based on proximity between two addresses, such as an insured's address or a user's address. You can also search for users by proximity on the administration user search page. This feature requires that ClaimCenter has the geocoding plugin enabled and that the ClaimCenter database contains geocoded addresses
- ClaimCenter catastrophe search and heat maps – ClaimCenter enables you to search for claims associated with a catastrophe and displays search results in a heat map on the **Catastrophe Search** page.
- ContactManager address book searches (proximity search of vendors) – ClaimCenter and ContactManager enable you to search for nearby service providers in the address book based on geographic proximity. This feature requires that:
 - ClaimCenter and ContactManager are installed.
 - ClaimCenter and ContactManager have the `GeocodingPlugin` enabled.

- ContactManager contains geocoded addresses in its database.

To support using geocoding with user assignment and searching, you must install and register a `GeocodePlugin` implementation for an external geocoding service in ClaimCenter. To support address book searches that use geocoding, you must install and register a `GeocodePlugin` implementation in both ClaimCenter and ContactManager.

IMPORTANT: To support address book searches with geocoding, configure and install both ClaimCenter and ContactManager, and then integrate the two applications. For details, see the *Contact Management Guide*.

What the geocoding plugin does

A `GeocodePlugin` implementation performs the following tasks.

Task	Required
Assigning latitude and longitude coordinates	•
Listing possible address matches	
Returning driving directions	
Finding an address from coordinates	
Finding maps for arbitrary addresses	

Synchronous and asynchronous calls to the geocoding plugin

From the perspective of ClaimCenter, calling a `GeocodePlugin` method is always synchronous. The caller waits until the method completes. For user-initiated requests, such as proximity searches, the user interface blocks until the plugin responds.

ClaimCenter and ContactManager also support a distributed system that enables multiple servers in the cluster to perform geocoding asynchronously from the application. This work queue system is especially useful after an upgrade with new addresses to geocode. You can use the geocoding work queue to geocode addresses in the background. You can use the work queue even if your application instance comprises a single server instead of a cluster of servers.

Note: The `Geocode` and `ABGeocode` work queues use only the `geocodeAddressBestMatch` method of the `GeocodePlugin` interface.

Using a proxy server with the geocoding plugin

To prevent ClaimCenter and the geocoding plugin from accessing external Internet services directly, you must use a proxy server for outgoing requests. If you use a proxy server for the geocoding plugin, you must configure the built-in Bing Maps implementation to connect with the proxy server, not the Bing Maps geocoding service.

The geocoding plugin only initiates communications with geocoding services. The plugin never responds to communications initiated externally from the Internet so you do not need a reverse proxy server to insulate the geocoding plugin from incoming Internet requests.

See also

- “Proxy servers” on page 37

Batch geocoding only some addresses

The `Address` entity has a property called `BatchGeocode`. The `Geocode` writer in ClaimCenter and the `ABGeocode` writer in ContactManager use `BatchGeocode` and other criteria to filter which addresses to pass to the plugin for geocoding.

- ContactManager verifies that the `BatchGeocode` property is true and the `GeocodeStatus` property is `none`.
- ClaimCenter verifies that the `BatchGeocode` property is true and the contact for the address has a subtype of `UserContact`.

If the address matches these conditions, the work queue writer passes the address to the plugin for geocoding.

Implementations of the `GeocodePlugin` ignore the `BatchGeocode` property. Callers of the plugin are responsible for determining which addresses to geocode. The `GeocodePlugin` geocodes any address it receives.

See also

- *Contact Management Guide*

Base implementation of the Bing maps geocoding plugin

ClaimCenter includes a fully functional and supported implementation of the `GeocodePlugin` to connect to the Microsoft Bing Maps Geocode Service.

BingMapsPluginRest plugin implementation class

The plugin implementation for geocoding addresses and driving directions is provided in the `BingMapsPluginRest` class. This class is registered in `GeocodePlugin.gwp` as the default plugin implementation.

In addition to username and password parameters, the plugin provides host name and version plugin parameters, all of which you can set in the registry `GeocodePlugin.gwp`.

There are general utility methods in the Gosu classes `GeocodingUtil` and `RoutingUtil` that `BingMapsPluginRest` calls to generate the requests. The appropriate address query parameters for geocoding requests are set from the `Address` entity, rather than concatenating the address fields into a general query parameter.

`RoutingUtil` provides two utility methods named `calculateSimpleDrivingRoute` that take different parameters for the start and end waypoints:

- One accepts `AbstractGeocodePlugin.LatLng` objects.
- One accepts `Address` entity instances.

The `BingMapUtils` class contains constants that are referenced throughout the plugin implementation and in its supporting classes, as well as helpers and the object mapper for Jackson deserialization.

Note: Guidewire recommends that you do not use polymorphic deserialization, or that you use it only with annotations on an individual class basis. Do not enable global default typing.

Geocoding and routing responses

The JSON geocoding or routing responses are deserialized by using Jackson, based on the `gw.api.plugin.geocode.impl.model` classes and `GeocodingResponse` and `RoutingResponse` classes.

Note: Guidewire recommends that you do not use polymorphic deserialization, or that you use it only with annotations on an individual class basis. Do not enable global default typing.

The `GeocodingResponse` and `RoutingResponse` classes implement the `Response` interface and generally contain:

- Status codes
- Descriptions
- A method to determine whether the response was a success
- A set of resources (location and a set of driving directions)

The Bing Map specification indicates there is a potential for multiple resource sets. In the `BingMapsPluginRest` class, the method `geocodeAddressWithCorrections` overrides the abstract plugin and returns multiple addresses up to the number specified in the `maxResults` parameter.

Geocoding request generators

General utility methods in the Gosu classes `GeocodingUtil` and `RoutingUtil` generate requests. If these default request generators do not match your requirements, you can add new methods to these two classes.

You can create new instances of `GeocodingRequest` and `RoutingRequest` either in the utility classes or in your implementation of `BingMapsPlugin`, and then set values to the defined query parameters. These classes extend `PendingResultBase`, which contains more general methods, such as setting query parameters. Not all the query parameters accepted by Bing Maps have been included in the base configuration, so you can also extend these classes to support other parameters.

Where applicable, parameters have value checking that will throw `IllegalArgumentException` exceptions if the conditions are violated. Also, requests contain a `validateRequest` method that is called when performing the request to get a Bing Maps JSON response. Generally, these requests take a `Context` object.

A new `Context` object with a user culture and application key is created per request in `BingMapsPluginRest`. It is possible to share `Context` objects across different requests. A `Context` object also has a `RequestHandler`, intended for use in the `Context` get or post methods to create the associated `PendingResult` object from the request. Note that the base implementation does not implement these methods. On a `PendingResult` execute, the actual request to Bing Maps is performed, and the response is then processed. Each request can run this method only once.

The base implementation provides a `RequestHandler` and associated `PendingResult` implementation, which is used in the default `Context` constructor, `HttpURLConnectionRequestHandler`. The handlers have unimplemented `handleGet` and `handlePost` methods that return an appropriate `PendingResult` object. Depending on the implementation, you can set values for connection, read, and socket timeout on the handlers. You can also create custom handlers and custom pending results.

A `Config` object can also be specified when constructing a request. This object specifies:

- REST API host name, with a default value of `https://dev.virtualearth.net/REST/`
- Version with a default value of v1
- Path, such as `/Routes`
- Resource path, such as `Driving`
- HTTP method for the request

Only `GET` is supported in the base configuration plugin implementation class for geocoding and routing.

Default `Config` objects are created if none are specified during request creation. In the plugin implementation class, if the plugin parameters `hostName` or `version` have been set in the `GeocodePlugin.gwp` registry editor, the `Config` objects for requests are overwritten with those values.

Deploy a geocoding plugin

About this task

Follow these steps to deploy a `GeocodePlugin` implementation:

Procedure

1. Implement the plugin interface in Gosu.

If you want to use a geocoding service other than the one supported by the built-in `GeocodePlugin` implementation, write your own implementation and register it in Guidewire Studio.

2. To support proximity searches of vendors in `ContactManager`, you must repeat the previous step in `ContactManager` Studio.
3. Register the plugin implementation in Studio:

- a) In the **Project** window in Studio, navigate to **Configuration > config > Plugins > registry**, and then open `GeocodePlugin.gwp`.
- b) In the pane on the right, clear the **Disabled** checkbox.
- c) In the **Gosu Class** text box, enter the name of the Gosu plugin implementation class that you want to use.
- d) Edit the **Parameters** table to specify parameters and values that your plugin implementation requires, such as security parameters for connecting to the external geocoding service, such as username and password.

4. To support proximity searching of vendors, which uses the ContactManager application, you must repeat the previous step in ContactManager Studio.
5. Enable the user interface for geocoding features by configuring parameters in the config.xml file:
 - a) In the **Project** window in Studio, navigate to **Configuration > config**, and then open the config.xml file.
 - b) For ClaimCenter, modify the UseGeocodingInPrimaryApp parameter. This parameter specifies whether ClaimCenter displays the user interface for proximity searches local to ClaimCenter in assignment and user search pages and pickers. For example:

```
<param name="UseGeocodingInPrimaryApp" value="true"/>
```
 - c) To support proximity searches of vendors in the user interface for geocoding, which uses the ContactManager application, set the UseGeocodingInAddressBook parameter. Set this parameter in each application. For example, to support geocoding in the user interface in both ClaimCenter and ContactManager, set the following parameter in both ClaimCenter and ContactManager.

```
<param name="UseGeocodingInAddressBook" value="true"/>
```

What to do next

See also

- *Administration Guide*

Writing a geocoding plugin

To use a geocoding service other than Microsoft Bing Maps Geocode Service, write your own GeocodePlugin implementation in Gosu and register your implementation class in Guidewire Studio.

The geocodeAddressBestMatch method is the only method required for a GeocodePlugin implementation to be considered functional. The other methods are for optional features of the GeocodePlugin.

The high level features and related plugin methods of the GeocodePlugin interface are listed below.

Feature	Plugin method	Required
Geocode an address	geocodeAddressBestMatch	•
List possible matches for an address	geocodeAddressWithCorrections pluginSupportsCorrections	
Retrieve driving directions between two addresses	getDrivingDirections pluginSupportsDrivingDirections pluginReturnsOverviewMapWithDrivingDirections pluginReturnsStepByStepMapsWithDrivingDirections	
Retrieve a map for an address	getMapForAddress pluginSupportsMappingByAddress	
Retrieve an address from a pair of geocode coordinates	getAddressByGeocodeBestMatch pluginSupportsFindByGeocode	
List possible addresses from a pair of geocode coordinates	getAddressByGeocode pluginSupportsFindByGeocodeMultiple	

Using the abstract geocode Java class

Guidewire provides a built-in, abstract implementation of the GeocodePlugin plugin interface called AbstractGeocodePlugin in the package gw.api.geocode. Your Gosu implementation of GeocodePlugin can extend

the `AbstractGeocodePlugin` Java class. Using the default behaviors of `AbstractGeocodePlugin` can save you work, particularly if you do not support all the optional features of the plugin.

If you use `AbstractGeocodePlugin` as the base class of your implementation, your Gosu class must implement the following methods.

- `geocodeAddressBestMatch`
- `getDrivingDirections`
- `pluginSupportsDrivingDirections`

You can add other interface methods to your Gosu class to support other optional features of the `GeocodePlugin`.

High-level steps to writing a geocoding plugin implementation

1. Write a new class in Studio that extends `AbstractGeocodePlugin`.

```
class MyGeocodePlugin extends AbstractGeocodePlugin {
```

2. Implement the required method `geocodeAddressBestMatch`.

The method accepts an address and returns a different address with latitude and longitude coordinates assigned.

3. To support driving directions, implement these methods.

- `pluginSupportsDrivingDirections` – Return `true` from this method to indicate that your implementation supports driving directions.
- `getDrivingDirections` – If your implementation supports driving directions, return driving directions based on a start address and a destination address that have latitude and longitude coordinates. Otherwise, return `null`.

4. If you want to support other optional features, such as getting a map for an address or getting an address from geocode coordinates, override additional methods. Identify to ClaimCenter that your plugin supports these features by implementing the methods with names that begin `pluginSupports`.

Geocoding an address

The `GeocodePlugin` interface has one required method, `geocodeAddressBestMatch`. The method takes an `Address` instance and returns a different `Address` instance. The address that the plugin returns has a `GeocodeStatus` value that indicates whether the geocoding request succeeded and how precisely the geocode coordinates match the incoming address. Valid values for `GeocodeStatus` include `exact`, `failure`, `street`, `postalcode`, or `city`. If the status is anything other than `failure`, the `Latitude` and `Longitude` properties in the returned address are correct for the returned address.

Your implementation of the `geocodeAddressBestMatch` method must not modify the incoming address instance for any reason, such as using data returned from the geocoding service. Instead, use the `clone` method on the incoming address to make a copy or create a new address instance by using the Gosu expression `new Address()`.

Set the properties on the cloned or new address from the values returned by the geocoding service, and use that address as the return value of your `geocodeAddressBestMatch` method.

The following example creates a new address and sets the geocoding status and the geocode coordinates.

```
a = new Address()
a.GeocodeStatus = GeocodeStatus.TC_EXACT
a.Latitude = 42.452389
a.Longitude = -71.375942
```

In a real implementation, your code assigns coordinate values obtained from the external geocoding service, not from numeric literals as the example shows.

Geocoding an address from the user interface

If ClaimCenter needs to geocode an address immediately, ClaimCenter calls one of the geocoding plugin methods. For example, to get the best match for a geocoding request, ClaimCenter calls the plugin method `geocodeAddressBestMatch`.

If you trigger geocoding from the user interface, geocoding is synchronous and the user interface blocks until the plugin returns the geocoding result. ClaimCenter has no built-in timeout between the application and the geocoding plugin. Your own geocoding plugin must encode a timeout so that it can abandon the call to the external service, throw a `RemoteException`, and resume the user interface operation.

Geocoding an address from a batch process

ClaimCenter geocodes addresses in the background by using batch processes that call the geocoding plugin.

Handling address clarifications for a geocoded address

Your plugin does not fully support address correction if you override only the `geocodeAddressBestMatch` method from `AbstractGeocodePlugin`. The `geocodeAddressBestMatch` method can provide address clarifications or leave some properties blank if the service did not use them to generate the coordinates. If the geocoding service modified properties on the submitted address, set those properties on the address that your plugin returns to the values of the modified properties.

To support address correction, override the `geocodeAddressWithCorrections` and `pluginSupportsCorrections` methods. You must also implement a PCF file for the user interface to display a list of multiple addresses from which the user selects the correct address.

Callers of the plugin must assume that blank properties in a returned address are intentionally blank. For example, certain address properties in return data might be unknown or inappropriate if the geocoding status is other than `exact`. If the geocode status represents the weighted center of a city, the street address might be blank because the returned geocode coordinates do not represent a specific street address. ClaimCenter treats the set of properties that the geocoding plugin returns as the full set of properties to show to the user or to log to the geocoding corrections table.

Sometimes a geocoding service returns variations of an address. For example, the street address “123 Main Street” might “123 North Main Street” and “123 South Main Street”, each with different geocode coordinates. The geocoding service might return both results so that a user can select the appropriate one. Some variations might be due to differences in abbreviations, such `Street` or `St.` Some services provide variants with and without suite, apartment, and floor numbers from addresses, or provide variants that contain other kinds of adjustments. For the `geocodeAddressBestMatch` method, return only the best match.

Supporting multiple address corrections with a list of possible matches

If your geocoding service can provide a list of potential addresses for address correction, implement the `geocodeAddressWithCorrections` method. Additionally, implement the `pluginSupportsCorrections` method and return `true` to indicate to ClaimCenter that your implementation supports multiple address corrections.

In contrast to the `geocodeAddressBestMatch` method, the `geocodeAddressWithCorrections` method returns a list of addresses rather than a single address. Both methods can return address corrections or clarifications, or leave some properties blank if they were not used to generate the coordinates. However, the system calls the `geocodeAddressWithCorrections` method if the user interface context can handle a list of corrections. For example, your user interface might support enabling a user to choose the intended address from a list of near matches.

The result list that your `geocodeAddressWithCorrections` method returns must be a standard `List<java.util.List>` that contains only `Address` entities. You declare this type of object in Gosu by using the generic syntax `List<Address>`.

If the geocoding service does not support multiple corrections, this method must return a one-item list that contains the results of a call to `geocodeAddressBestMatch`. If you base your implementation on the built-in `AbstractGeocodePlugin` class, the abstract class implements this behavior for you.

Geocoding error handling

If your plugin implementation fails to connect to the external geocoding service, throw the exception `java.rmi.RemoteException`. Rather than setting the geocode status of an address to none, you can throw an exception if the error is retryable.

Getting driving directions

ClaimCenter and ContactManager optionally can display driving directions and other travel information. You can support this feature by implementing the following methods on the `GeocodePlugin` interface.

- `getDrivingDirections` – Get driving directions based on a start address and a destination address, as well as a switch that specifies miles or kilometers.
- `pluginSupportsDrivingDirections` – Return true from this simple method.

Driving directions are enabled by default if you have geocoding enabled in the user interface. To disable driving directions even if geocoding is enabled, you must edit the relevant PCF files.

ClaimCenter does not require that the driving directions request be handled by the same external service as geocoding requests. The plugin could contact different services for the two types of requests.

The `GeocodePlugin` interface provides a single method called `getDrivingDirections` to support driving directions. ClaimCenter calls the geocoding plugin method `getDrivingDirections` once for each user request for driving directions. From a programming perspective, the method and the request are synchronous because the method must not return until the request is complete. However, from a user perspective the request might seem asynchronous because ClaimCenter can request multiple driving direction requests without showing them to the user immediately. For example, ClaimCenter requests directions from one insured's home address to nine different auto repair shops. The user can then choose the map for one of the shop locations.

The `getDrivingDirections` method takes two `Address` entities. The method must send these addresses to a remote driving directions service and return the results. If driving time and a map showing the route between the two addresses are available, the method also returns these two values.

Address properties already include values for latitude and longitude before calling the plugin. Because some services use only the latitude and longitude, driving directions can be to or from an inexact address such as a postal code rather than exact addresses.

The plugin must return a `DrivingDirections` object that encapsulates the results. This class is in the `gw.api.contact` package. To create a `DrivingDirections` object, you have two options.

- You can directly create a new driving directions object.

```
var dd = new DrivingDirections()
```

- You can use a helper method to initialize the object.

Retrieving overview maps

If your geocoding or routing service supports overview maps, first implement the method `pluginReturnsOverviewMapWithDrivingDirections` and have it return true.

Next, set the following properties on the driving directions object.

- `MapOverviewUrl` – a URL of the overview map shows the entire journey as a `MapImageUrl` object, which is a simple object containing two properties.
 - `MapImageUrl` – A fully-formed and valid URL string.
 - `MapImageTag` – The text of the best HTML image element, an HTML `` tag, that properly displays this map. This text can include, for example, the `height` and `width` attributes of the map if they are known.
- `hasMapOverviewUrl` – Set to true if there is a URL of the overview map shows the entire journey

To see how map data is used in the ClaimCenter user interface, open the PCF file `AddressBookDirectionsPopup.pcf`.

Adding segments of the journey with optional maps

If you want to provide actual driving directions, you must also add driving direction elements that represent the segments of the journey.

Each `DrivingDirections` object provides various properties that relate to the entire journey. Additionally, `DrivingDirections` has an array of `DrivingDirectionsElem` objects that provide a list of journey segments. Each object in the array represents one segment, such as "Turn right on Main Street and drive 40 miles". Many properties are set automatically if you use `createPreparedDrivingDirections` as described previously in the description of how to initialize a `DrivingDirections` object.

For each new segment, you do not need to create `DrivingDirectionsElem` directly. Instead call the `addNewElement` method on the `DrivingDirections` object.

```
drivingdirections.addNewElement(String formattedDirections, Double distance, Integer duration)
```

The `formattedDirections` object can represent either of the following items.

- A discrete stage in the directions, such as "Turn left onto I-80 for 10 miles."
- A note or milestone not corresponding to a stage, such as "Start trip."

The exact format and content of the textual description depends on your geocoding service. It can include HTML formatting.

If your service supports multiple individual maps other than the overview, repeatedly call the method `addNewMapURL(urlString)` on the driving directions object to add URLs for maps. There is no requirement for the number of maps to match the number of segments of the journey. The position in the map URL list does not have a fixed correspondence to a segment number. However, always add map URLs in the expected order from the start address to the end address.

If you add individual maps, also implement the method `pluginReturnsStepByStepMapsWithDrivingDirections` and have it return `true`.

Extracting data from driving directions in PCF files

If you extract information from `DrivingDirections` in PCF code or other Gosu code, there are properties you can extract, such as `TotalDistance`, `TotalTimeInMinutes`, `GCDistance`, and `GCDistanceString`. Methods with GC in the name refer to the great circle and great circle distance, which is the distance between two points on the surface of a sphere. It is measured along a path on the surface of the Earth's curved 3-D surface, in contrast to point-to-point through the Earth's interior.

Note: In Gosu, the address entity contains utility methods for calculating great circle distances. For example, `address.getDistanceFrom(latitudeValue, longitudeValue)`.

The `description` properties from the start and end addresses are also copied into the `Start` and `Finish` properties on `DrivingDirections`.

Error handling in driving directions

If your plugin implementation fails to connect to the external geocoding service, throw the exception `java.rmi.RemoteException`.

Getting a map for an address

If your geocoding service supports getting a map from an address, first implement the `pluginSupportsMappingByAddress` method and return `true`. Next, implement the method `getMapForAddress`, which takes an `Address` and a unit of distance, which is either miles or kilometers.

Your `getMapForAddress` method must return a map image URL in a `MapImageUrl` object. This object is a wrapper of a `String` for a map URL.

The following code demonstrates a simple (fake) implementation.

```
override function getMapForAddress(address: Address, unit: UnitOfDistance) : MapImageUrl {
    var i = new MapImageUrl()
    i.MapImageUrl = "http://myserver/mapengine?lat=3.9&long=5.5"
    return i;
}
```

Getting an address from coordinates (reverse geocoding)

Some geocoding services support *reverse geocoding*, getting an address from latitude and longitude coordinates. If your service supports it, you can override the following methods in `AbstractGeocodePlugin` to implement reverse geocoding.

Note: ClaimCenter supports two types of reverse geocoding: return a single address, and return multiple addresses. You can support one or both. Implementing the following methods is required if you want to support all mapping features in ClaimCenter.

To implement single-result reverse geocoding, first implement the method `pluginSupportsFindByGeocode` and have it return `true`. Next, implement the `getAddressByGeocodeBestMatch` method, which takes a latitude coordinate and a longitude coordinate. Return an address with as many properties set as your geocoding service provides.

To implement multiple-result reverse geocoding, first implement the method `pluginSupportsFindByGeocodeMultiple` and have it return `true`. Next, implement the `getAddressByGeocode` method, which takes a latitude coordinate, a longitude coordinate, and a maximum number of results. If the maximum number of results parameter is zero or negative, this method must return all results. As with the methods used for multiple-result geocoding, this method returns a list of `Address` entities, specified with the generics syntax `List<Address>`.

Geocoding status codes

Geocoding services typically provide a set of status codes to indicate what happened during the geocoding attempt. Even if the external geocoding service returns latitude and longitude coordinates successfully, it is useful to know how precisely those coordinates represent the location of an address.

- The coordinates represent an exact address match.
- If the service could not find the address or the address was incomplete, the coordinates could identify the weighted center of the postal code or city.

The status codes `exact`, `street`, `postalcode`, and `city` indicate the precision with which the `Latitude` and `Longitude` properties identify the global location of an address.

Additionally, the status code `failure` indicates that geocoding failed, making any values in the `Latitude` and `Longitude` properties of the address unreliable. The status code `none` indicates that an address has not been geocoded since it was created or last modified, which also means that `Latitude` and `Longitude` are unreliable.

List of geocoding status codes

The status values must be values from the `GeocodeStatus` type list, described in the following table.

Geocode status code	Description
none	No attempt at geocoding this address occurred. This value is the default geocoding status for an address. If you experience an error that must retrigger geocoding later, rather than setting this value, you can throw an exception. When an address is modified, ClaimCenter sets the address to this status, which indicates that the address has not been geocoded since it was last modified.

Geocode status code	Description
failure	An attempt at geocoding this address was made but failed completely. If an address could not be geocoded, use this code. Do not use this code for an error that is retryable, such as a network failure. If you experience an error that must retrigger geocoding later, throw an exception instead.
exact	This address was geocoded successfully, and the service provider indicates that the supplied geocode coordinates match exactly the complete address.
street	This address was geocoded successfully, and the service provider indicates that the supplied geocode coordinates match the street but not the complete address. This status can be more or less precise than <code>postalcode</code> , depending on the complete address and the length of the street.
postalcode	This address was geocoded successfully, and the service provider indicates that the supplied geocode coordinates match the postal code, but not the complete address. The meaning of a postal code match depends on the geocoding service. A geocoding service can use the geographically weighted center of the area, or it can use a designated address within the area, such as a postal office.
city	This address was geocoded successfully, and the service provider indicates that the supplied geocode coordinates match the city, but not the complete address. The meaning of a city match depends on the geocoding service. A geocoding service can use the geographically weighted center of the city, or it can use a designated address, such as the city hall.

IMPORTANT: The `GeocodeStatus` type list is final. You cannot extend it with type codes of your own.

Reinsurance integration

Reinsurance is insurance risk transferred to another insurance company for all or part of an assumed liability. You can integrate external systems with ClaimCenter to retrieve reinsurance information about policies on claims.

Reinsurance plugin

ClaimCenter provides the `IReinsurancePlugin` to synchronize reinsurance information for policies on claims in ClaimCenter with current reinsurance information from an external system.

The default configuration of ClaimCenter provides the following built-in implementations.

- `ReinsuranceDemoPlugin.gs` – Simulates integration with an external system by returning sample data to ClaimCenter.
- `PCReinsurancePlugin.gs` – Integrates PolicyCenter and Reinsurance Management with ClaimCenter through the `RICoverageAPI` web service that PolicyCenter publishes.

Enabling the reinsurance plugin for production

About this task

In the base configuration, the `IReinsurancePlugin` is enabled and uses the `ReinsuranceDemoPlugin` implementation class. This demonstration class can be useful when you first begin to work with reinsurance in ClaimCenter. It demonstrates the default reinsurance features of ClaimCenter. You must not use this implementation in a production setting.

Procedure

1. At a command prompt, navigate to the `ClaimCenter` root directory and run the following command to start Guidewire Studio.

```
gwb studio
```
2. In the Studio **Project** window, navigate to **configuration > config > Plugins > registry**, and open the file `IReinsurancePlugin.gwp`.
3. Click the **Enabled** check box to enable the plugin. If a message asks whether you want to create a copy in the current module, click **Yes**.
4. In the **Gosu Class** field, specify the following class name.

`gw.plugin.policy.reinsurance.pc1000.PCReinsurancePlugin`

Encryption integration

Some data model properties can be encrypted before ClaimCenter stores them in the database. Encryption enables the protection of sensitive data, such as bank account information or personal data, by storing it in a non-plaintext format. ClaimCenter supports the encryption of `String` properties. If you need to encrypt other data types, you must convert them to `String` values.

You can implement a custom encryption algorithm in the `IEncryption` plugin. The encryption ID of the plugin implementation identifies the encryption algorithm to ClaimCenter.

IMPORTANT: If you make any change to the algorithm that causes encrypted values to differ from the previous implementation, you must change the encryption ID. If you do not change the encryption ID, previously encrypted values in the database are not decrypted correctly and appear corrupted.

Using encrypted properties

From Gosu, encrypted fields retrieved from the database are decrypted and appear as plaintext. Similarly, the plaintext values of encrypted properties are encrypted before they are saved to the database.

All encryption and decryption occurs within ClaimCenter automatically whenever the application reads or writes entity instances in the database. Gosu rules, web services, and messaging plugins operate on plaintext, or decrypted, strings without special code to manage their encryption and decryption.

Adding or removing encrypted properties

If you add or remove encrypted properties in the data model, the upgrader automatically runs on server startup to update the main database to the new data model. Similarly, if you change the value of the `encryption` column parameter on a column in the data model, the upgrader encrypts or decrypts the values in the database to match the change in the data model.

The upgrader does not upgrade claim snapshots or archive databases during the normal upgrade process.

Duplicate-check search template and encryption

In the base configuration, the ClaimCenter duplicate-check search template supports encryption.

Setting encrypted properties

You can change the encryption settings for a column in data model files by overriding the column information. You can set encryption only for text column types, `shorttext`, `mediumtext`, and `longtext`. Encryption is unsupported on other

types, including binary types (`varbinary`) and date types. A column of type `longtext` is a large character object (`CLOB`).

IMPORTANT: Encrypting a `longtext` column might require a large amount of memory.

To mark a column as encrypted, add a `<columnParam>` element in the appropriate `<column>` element. In the new element, set the `name` attribute to `encryption` and the `value` attribute to `true`. The following example encrypts the property `TestProperty`.

```
<column name="TestProperty" type="shorttext" nullok="true" desc="Example of an encrypted column" >
<columnParam name="encryption" value="true"/>
</column>
```

If the value of a user interface widget is set to a property expression and the property is encrypted, the widget displays the data as visually masked for privacy. Note that the application does not display an input mask if the expression is a method invocation rather than a property expression.

Encryption of denormalized columns

Encryption is prohibited on any property that uses a secondary column to support case-insensitive search. ClaimCenter prevents you from encrypting properties that have this denormalized column. For example, the contact property `LastNamesDenorm` is a denormalized column that mirrors the `LastName` property. Therefore, the `LastName` property cannot be encrypted.

Encryption of date columns

Date columns cannot be directly encrypted. If you store a date in the database as text, you can encrypt the field. To access the value as a date, you would have to convert the text value to a `Date` object. Similarly, you would have to convert the `Date` object to a text value before storing it in the database. You cannot do searches, have useful database indexes, or sort records on that column because that column has encrypted values in the database.

If the limitations of this approach are acceptable, you can implement conversion to and from a `Date` object by using a Gosu enhancement. For example, suppose you add a date-of-birth column to the `Contact` entity type. Add a column called `Dob_Ext` with type `varchar` with a size `columnParam` of 10 characters. Add an encryption `columnParam` with the value `true`. Create a Gosu enhancement with a property setter and getter for a new property called `ContactDOB`.

```
property set ContactDOB(dob : Date) {
    if (dob != null) {
        this.Dob_Ext = new SimpleDateFormat("yyyy-MM-dd").format(dob)
    }
}

property get ContactDOB() : Date {
    if (this.Dob_Ext != null) {
        return new SimpleDateFormat("yyyy-MM-dd").parse(this.Dob_Ext)
    }
    return null
}
```

Update the date-format string in the example to match your preferred date format in the text field.

Querying encrypted properties

Comparisons using the query builder API succeed if you compare encrypted values in the database to plaintext values in the application. For example, a valid comparison occurs when you query the database for a taxpayer ID by comparing a plaintext value entered by a user to encrypted values in the database. ClaimCenter encrypts the plaintext value and the query builder API passes that encrypted value to the database. The database compares the encrypted value of the plaintext string to the encrypted values in the taxpayer ID column.

Restrictions on querying encrypted properties

The query builder API restricts comparisons to encrypted values in the database to equality comparisons only. You can use either “equals” or “not equals” operators to compare a plaintext value with the encrypted database values. You

cannot use relative operators, such as “greater than” or “less than.” You can use the query builder API to make an equality comparison of an encrypted column to one of the following items.

- A Gosu expression that evaluates to a `String`
- Another encrypted column

You cannot use the query builder API to compare an encrypted column to a plaintext column because the database cannot encrypt a plaintext column value to make the comparison valid. Similarly, the database cannot decrypt the values in the encrypted column to plaintext values.

You cannot use the query builder API to sort records on an encrypted column because the values in the database are encrypted and therefore do not sort in a meaningful way.

Examples of querying encrypted properties

For the examples, assume the following context.

- `Claim.SecretVal` and `Claim.SecretVal2` are encrypted properties.
- `Claim.ClearVal` and `Claim.ClearVal2` are plaintext properties.
- `tempStringValue` is a local variable that contains a plaintext `String` value.

The following comparison methods are valid.

```
// Comparison of an encrypted column to a String literal or variable
query.compare(Claim#SecretVal, Equals, "123")
query.compare(Claim#SecretVal, NotEquals, "123")
query.compare(Claim#SecretVal, Equals, tempStringValue)
query.compare(Claim#SecretVal, NotEquals, tempStringValue)

// Comparison of an encrypted column to another encrypted column
query.compare(Claim#SecretVal, Equals, q.getColumnRef("SecretVal2"))
query.compare(Claim#SecretVal, NotEquals, q.getColumnRef("SecretVal2"))

// Comparison of plaintext column with other plaintext values
query.compare(Claim#ClearVal, Equals, q.getColumnRef("ClearVal2"))
query.compare(Claim#ClearVal, NotEquals, q.getColumnRef("OtherVal2"))
```

The following comparisons are not valid. Although the SQL command that ClaimCenter sends to the database is syntactically correct, the comparison of a plaintext value and an encrypted value is not meaningful. Similarly, using a relative operator to compare an encrypted value against the encrypted values in the database is not meaningful.

```
// Comparison of an encrypted column to a plaintext column
query.compare(Claim#SecretVal, Equals, q.getColumnRef("ClearVal"))

// Comparison of a plaintext column to an encrypted column
query.compare(Claim#ClearVal, NotEquals, q.getColumnRef("SecretVal"))

// Comparison of an encrypted column using a relative operator
query.compare(Claim#SecretVal, GreaterThan, "123")
```

The following ordering request is not valid. Although the SQL command that ClaimCenter sends to the database is syntactically correct, sorting rows by an encrypted column is not meaningful.

```
query.select().orderBy(QuerySelectColumns.path(Paths.make(Claim#SecretVal)))
```

Sending encrypted properties to other systems

Sending encrypted properties to external systems

From Gosu, encrypted fields retrieved from the database are decrypted and appear as plaintext. If you must avoid sending this information as clear text to external systems, you must design your own integrations to use a secure protocol or to encrypt the data yourself.

For example, use the plugin registry and use your own encryption plugin to encrypt and decrypt data across the wire in your integration code. Create classes to contain your integration data for each integration point. You can make some properties contain encrypted versions of data that require extra security between systems. Such properties do not need

to be database-backed. You can implement enhancement properties on entities that dynamically return the encrypted version. If your messaging layer uses encryption, such as SSL/HTTPS, or is on a secure network then additional encryption might not be necessary. It depends on the details of your security requirements.

From Gosu, any encrypted fields appear as plaintext. Carefully consider security implications of any integrations with external systems that send properties which are encrypted in the database.

Sending encrypted properties to other InsuranceSuite applications

For communication between Guidewire InsuranceSuite applications, the built-in integrations do not encrypt any encrypted properties during the web services interaction. This affects the following integrations.

- ClaimCenter and ContactManager
- ClaimCenter and PolicyCenter

To add additional security, you must customize the integration to use HTTPS or add additional encryption of properties sent across the network.

Setting up encryption

To set up encryption, you must register a startable plugin class that implements the `IEncryption` interface. Then, you provide a value for the `CurrentEncryptionPlugin` environment parameter. This value is the value of the `EncryptionIdentifier` property of the encryption plugin implementation.

When you next start the application, the upgrade process uses the encryption plugin to encrypt any encrypted properties that are stored in the database. Similarly, if you later remove the value for the `CurrentEncryptionPlugin` environment parameter, the upgrade process uses the previously defined encryption plugin to decrypt any encrypted properties that are stored in the database.

Changing the encryption plugin causes the upgrade process to change the values in encrypted fields in the database. This process requires a full table scan of all tables that have encrypted fields. Depending on the size of your database tables and the number of encrypted fields, this process can take a long time. You need to test how long this process takes by using a realistic test database before you make the change to your production database.

IMPORTANT: If you make any change to the algorithm that causes encrypted values to differ from the previous implementation, you must change the encryption ID. If you do not change the encryption ID, previously encrypted values in the database are not decrypted correctly and appear corrupted.

See also

- “[Changing your encryption algorithm](#)” on page 269
- [Configuration Guide](#)

Defining the encryption algorithm

You can implement your own encryption algorithm as an implementation of the `IEncryption` plugin. You must implement the plugin to properly encrypt your data with your own algorithm. ClaimCenter provides an example implementation of this plugin, `gw.plugin.encryption.impl.PBEEncryptionPlugin`, in the base configuration. Enable the plugin if you want to test the use of encryption.

Note that ClaimCenter does not support using a unique salt for every value that is encrypted.

Externalizing encryption

As best practice, externalize key and salt credential values for your encryption algorithm.

Writing your encryption plugin

Write and implement a class that implements the `IEncryption` plugin interface. Its responsibility is to encrypt and decrypt data with one encryption algorithm. This plugin encrypts and decrypts `String` values to other `String` values. This plugin does not support using a unique salt for every call to encrypt a value.

The base configuration of ClaimCenter does not include an example of a production-level encryption plugin.

To encrypt, implement an `encrypt` method that takes an unencrypted `String` and returns an encrypted `String`, which might be a different length from the original. If you want to use strong encryption and are permitted to use such encryption legally, you can do so. Because the `encrypt` method returns a `String` value, you typically use base-64 encoding to convert the encrypted value to a valid `String` value.

To decrypt, implement a `decrypt` method that takes an encrypted `String` and returns the original unencrypted `String`. If you use base-64 encoding in your `encrypt` method, you must decode the `String` value before decrypting it.

You must also specify the maximum length of the encrypted string by implementing the `getEncryptedLength` method. Its argument is the length of the decrypted data. It must return the maximum length of the encrypted data. ClaimCenter primarily uses the encryption length at application startup time during upgrades. During the upgrade process, the application must determine the required length of encrypted columns. If the length of the column must increase to accommodate inflation of encrypted data, this method provides ClaimCenter with the necessary information for how far to increase space for the database column.

To uniquely identify your encryption algorithm, your plugin must return an encryption ID. Implement a `getEncryptionIdentifier` method to return a unique identifier for your encryption algorithm. The identifier tracks encryption-related change control and is exposed to Gosu as the property `EncryptionIdentifier`.

```
override property get EncryptionIdentifier() : String {  
    return "ABC:DES3"  
}
```

The encryption ID must be unique among all encryption plugins in your implementation. The application decides whether to upgrade the encryption data with a new algorithm by comparing the following encryption IDs.

- The encryption ID of the current encryption plugin
- The encryption ID associated with the database last time the server ran

Detecting accidental duplication of encryption or decryption

You can mitigate the risk of accidentally encrypting already-encrypted data if you design your encryption algorithm to detect whether the property data is already encrypted.

For example, suppose you put a known series of special characters that could not appear in the data both before and after your encrypted data. You can now detect whether data is already encrypted or unencrypted and handle the situation appropriately.

Example encryption plugin

ClaimCenter provides an example encryption plugin implementation, `gw.plugin.encryption.impl.PBEEncryptionPlugin`, in the base configuration. This class uses classes in the `javax.crypto` package hierarchy to perform encryption and decryption. This class uses the Java encryption registry to access the encryption implementation. If you write your own security package and register it, you can use the `PBEEncryptionPlugin` to test your package.

This class is a useful example, but is not sufficient for a production environment. This class does not externalize the key or the salt credentials. You can extend the class for your own encryption plugin implementation or write a new class.

Plugin parameters

This plugin class uses the following plugin parameters:

identifier

The encryption identifier for this implementation of the `IEncryption` plugin interface

The server maps this identifier to the plugin implementation. Do not change the identifier after the plugin has been used.

iterationCount

The number of times to iterate over the key phrase to produce the key

You can use the default value of 19.

key

A long phrase to use as the password for the encryption

Change the value of this parameter.

PBE.digest

The cryptographic hash function to use to generate the hash key for the data

The default value is `SHA1`.

PBE.encryption

The algorithm to use to generate the encrypted data

The default value is `DESede`.

salt

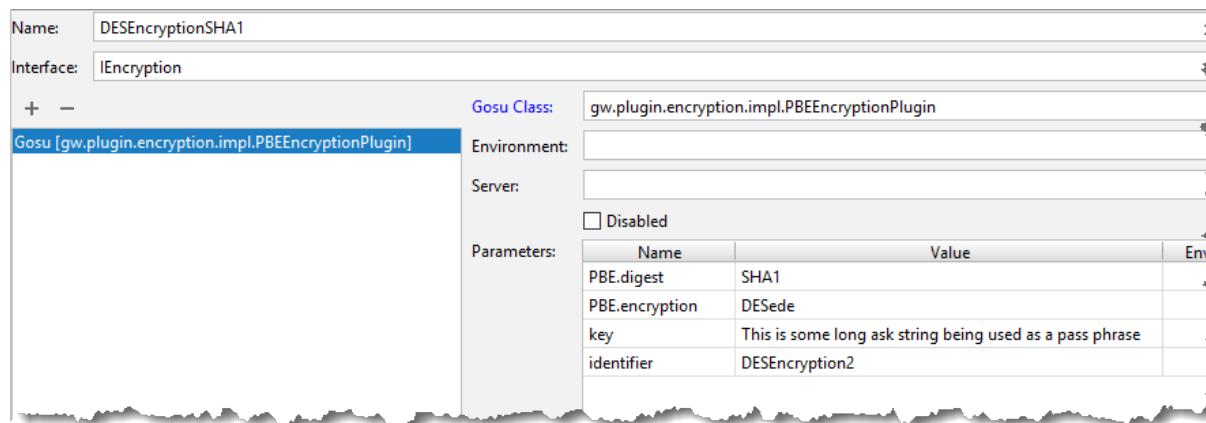
The salt value to use with the cryptographic hash function

You can use the default value for this parameter.

The plugin parameters for this class affect its behavior, as shown in the following examples. Both examples set values for the `key` and `identifier` parameters.

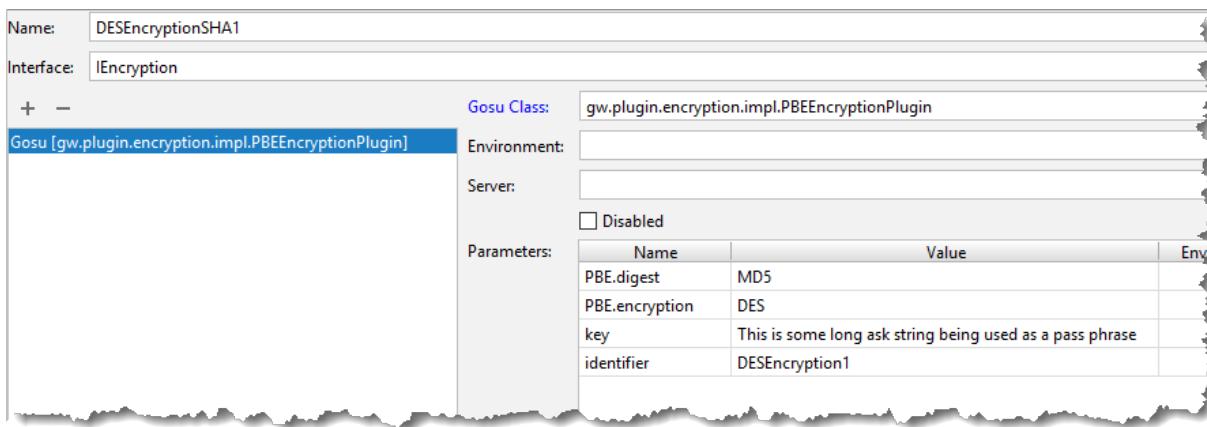
[Encryption plugin using SHA1 and DESede](#)

The following plugin specifies the `PBE.digest` and `PBE.encryption` parameters even though the values are the same as the defaults in the `PBEEncryptionPlugin` class. Specifying the parameters ensures that users of this plugin can see the mechanisms that the plugin uses.



[Encryption plugin using MD5 and DES](#)

The following plugin specifies the `PBE.digest` and `PBE.encryption` parameters to values that are different from the defaults in the `PBEEncryptionPlugin` class.



Changing your encryption algorithm

You can register any number of `IEncryption` plugins. For an original upgrade of your database to a new encryption algorithm, you register two implementations at the same time.

You might register more than two implementations if you have encrypted claim snapshots in archive databases.

However, only one encryption plugin is the current encryption plugin. The `config.xml` configuration parameter `CurrentEncryptionPlugin` controls this setting. It specifies which encryption plugin, among potentially multiple implementations, is the current encryption algorithm for the main database. Set the parameter to the plugin name, not the class name nor the encryption ID, for the current encryption plugin.

The server uses an internal lookup table to map all previously used encryption IDs to an incrementing integer value. This value is stored with database data. Internally, the upgrader manages this lookup table to determine whether data needs to be upgraded to the latest encryption algorithm. Do not attempt to manage this table directly. Instead, ensure every encryption plugin returns its appropriate encryption ID, and ensure `CurrentEncryptionPlugin` specifies the correct plugin name.

The encryption ID of a plugin implementation is not its plugin name nor its class name. The server relies on the encryption ID saved with the database and the encryption ID of the current encryption plugin to identify whether the encryption algorithm changed.

The following list shows the requirements if you change encryption algorithms.

- All encryption plugins must return their appropriate encryption IDs correctly.
- All encryption plugins must implement `getEncryptedLength` correctly.
- You must set `CurrentEncryptionPlugin` to the correct plugin name.

During server startup, the upgrader checks the encryption ID of data in the main database. The server compares this encryption ID with the encryption ID associated with the current encryption plugin. If the encryption IDs are different, the upgrader decrypts encrypted fields with the old encryption plugin, found by its encryption ID. Next, the server encrypts the fields to be encrypted with the new encryption plugin, found by its plugin name as specified by the parameter `CurrentEncryptionPlugin`.

On server startup, the upgrade process updates encryption settings in the database if any of the following events occurs.

- Add encrypted properties.
- Remove encrypted properties.
- Add an encryption plugin for the first time.
- Change the encryption algorithm. The server looks for a changed value for the encryption ID of the current encryption plugin.

Encryption changes with snapshots

However, the upgrade process does not upgrade claim snapshots during the normal upgrade process.

After startup, a work queue runs. Eventually that work queue converts all claim snapshots to use the current encryption settings. First the work queue decrypts if necessary then encrypts with the current encryption algorithm.

Additionally, claim snapshots can upgrade on demand for the user interface. Suppose the work queue is running but is not complete and did not upgrade some claim snapshots. If a user views one of those claim snapshots, ClaimCenter immediately upgrades the encryption in the snapshot as part of accessing the snapshot.

To control the number of claim snapshots the work queue upgrades at one time, set the `SnapshotEncryptionUpgradeChunkSize` configuration parameter. Set that parameter to the number of claims to upgrade each time the worker runs. Typically the worker runs once a day, but it is configurable.

To force the work queue to upgrade all snapshots all at once, set the `SnapshotEncryptionUpgradeChunkSize` parameter to 0. Next, after the usual database upgrade completes, manually run the Encryption Upgrade work queue writer from the tools page. However, before setting `SnapshotEncryptionUpgradeChunkSize` to 0 to process all snapshots, consider carefully the implications. Depending on the workload, this work queue can use server resources for a long time. In many production environments, good practice is to convert one chunk of snapshots every night at a time when server usage is low, such as 1:00 a.m.

Special issues for changing your encryption algorithm

For claim snapshots, the encryption ID used to encrypt the snapshot is saved in the claim snapshot with each encrypted field. The upgrader does not upgrade claim snapshots or archive databases during the normal upgrade process.

If you change encryption algorithms, you must continue to register one encryption plugin implementation for every encryption algorithm that might be referenced by the encryption ID in any claim snapshots. Ensure that all encryption plugin implementations return the correct encryption ID.

Change your encryption algorithm

About this task

The following procedure describes how to change your encryption algorithm. It is extremely important to follow it exactly and very carefully. Failure to perform this procedure correctly risks data corruption.

Procedure

1. Shut down your server.
2. Register a new plugin implementation of the `IEncryption` plugin for your new algorithm.
Studio prompts you for a plugin name for your new implementation.
3. Name the plugin appropriately to match the algorithm.
For example: `encryptDES3`
4. Be sure your plugin returns an appropriate and unique encryption ID. Name it appropriately to match the algorithm. For example, "encryptDES3."
5. Set the `config.xml` configuration parameter `CurrentEncryptionPlugin` to the plugin name of your new encryption plugin.
Do not delete your older encryption plugins from the registry. ClaimCenter uses these plugins to decrypt any encrypted data in claim snapshots in archived claims.
6. Start the server.

Results

The upgrader uses the previous encryption plugin to decrypt your data and then encrypts the data with the new algorithm.

Installing your encryption plugin

After you write your implementation of the `IEncryption` plugin, you must register it. You can register multiple `IEncryption` plugin implementations if you need to support changing the encryption algorithm.

Enable your encryption plugin implementation

Procedure

1. In Guidewire Studio, create a new class that implements the `IEncryption` plugin interface.

Be certain that your `EncryptionIdentifier` method returns a unique encryption identifier. If you change your encryption algorithm, it is critical that you change the encryption identifier for your new implementation.

Guidewire strongly recommends that you set the encryption ID for your current encryption plugin to a name that describes or names the algorithm itself. For example, "encryptDES3."

2. In the **Project** window in Studio, navigate to **configuration > config > Plugins > registry**.

3. Right-click **registry**, and choose **New > Plugin**.

4. Studio prompts you to name your new plugin.

You can have more than one registered implementation of an `IEncryption` plugin interface. The name field must be unique among all plugins. This name is the plugin name and is particularly important for encryption plugins.

5. In the interface field, type `IEncryption`.

6. Edit standard plugin fields in the Plugins editor in Studio.

7. In `config.xml`, set the `CurrentEncryptionPlugin` parameter to the plugin name.

The `CurrentEncryptionPlugin` parameter specifies the encryption plugin that is the current encryption algorithm for the main database. Specify the plugin name (not the class name).

If the `CurrentEncryptionPlugin` parameter is missing or specifies an implementation that does not exist, the server does not start.

8. Start the server.

If the upgrade tool detects data model fields that are marked as encrypted, but the database contains unencrypted versions, the upgrade tool encrypts the field in the main database by using the current encryption plugin.

Encryption features for staging tables

If you need to use staging tables to import records and any data has encrypted properties, you must encrypt those properties before importing them into operational tables. The table import tools provide options to encrypt any encrypted fields in staging tables before import. These tools encrypt only the columns that are marked as encrypted in the data model.

During upgrades, be careful about the order in which you make data model modifications. Before beginning staging table import work, do any data model changes including changing encryption settings. Next, let the server upgrade the database. After you modify the data model, during server launch, ClaimCenter automatically encrypts data in the operational tables as part of upgrade routines. During upgrade, ClaimCenter deletes all staging tables.

Guidewire strongly recommends that you encrypt your staging table data after you complete all work on your data conversion tools and your data passes all integrity checks. Note that staging table integrity checks never depend on the encryption status of encrypted properties. You can run integrity checks independent of whether the encrypted properties are currently encrypted.

ClaimCenter provides the following ways to encrypt your data.

- Asynchronously with the command line tool `table_import` with the options `-encryptstagingtbls -batch`.
- Asynchronously with web services using the `TableImportAPI` web service method `encryptDataOnStagingTablesAsBatchProcess`. The method takes no arguments and returns a process ID. Use

the returned ID to check the status of the encryption batch process or to terminate the encryption batch process. Use the ID with methods on the web service `MaintenanceToolsAPI`.

Encrypt any encrypted properties in staging tables

Procedure

1. In Guidewire Studio, write and register an encryption plugin.
2. Upgrade your existing production data after your data model change.
3. Perform integrity checks until all checks pass.

During development of conversion programs that populate the staging tables, you might need to repeatedly run integrity checks and modify your code.

4. Use one of the web service APIs or command line tools that encrypt columns in staging tables.

IMPORTANT: If the encryption process succeeds, do not run the process a second time because doing so would corrupt your data by encrypting already-encrypted data.

If the encryption process fails, no changes are committed to the database. All changes occur in a single database transaction. You can safely repeat the encryption process after all errors are fixed.

5. After the encryption process succeeds, use regular staging table loading web service APIs or command prompt tools.

For example, use the `table_import` command-prompt tool or call the web service method `integrityCheckStagingTableContentsAndLoadSourceTables`. These actions run integrity checks, which work with encrypted properties.

Free-text search integration

ClaimCenter free-text search is an alternative to database search that can return results faster for certain search requests than database search. Free-text search depends on an external full-text search engine, the Guidewire Solr Extension. Free-text search provides two plugins that connect free-text search in ClaimCenter with the Guidewire Solr Extension.

See also

- *Configuration Guide*
- *Installation Guide*

Overview of free-text search plugins

ClaimCenter free-text search depends on two Guidewire plugins that connect ClaimCenter to a full-text search engine. The search engine is a modified form of Apache Solr in a special distribution known as the *Guidewire Solr Extension*. The Guidewire Solr Extension runs in a different instance of the application server than the instance that runs your ClaimCenter application.

The free-text search plugin interfaces are:

ISolrMessageTransportPlugin

Called by the Indexing System rules in the Event Fired ruleset whenever contacts on claims change. The plugin extracts the changed data and sends it in an indexing document to the Guidewire Solr Extension for loading and incremental indexing.

ISolrSearchPlugin

Called by the **Search > Contacts** search screen to send users' criteria to the Guidewire Solr Extension and receive the search results.

ClaimCenter provides the plugin implementations and the Guidewire Solr Extension software. Do not connect the free-text plugins to your own installation of Apache Solr.

IMPORTANT: Guidewire does not support replacing the plugin implementations that ClaimCenter provides with custom implementations to other full-text search engines. Guidewire supports connecting the free-text plugins only to a running instance of the Guidewire Solr Extension.

Connecting the free-text plugins to the Guidewire Solr Extension

If you configure free-text search for external operation, the free-text plugins connect to the Guidewire Solr Extension through the HTTP protocol. The plugin implementations obtain the host name and port number for the Guidewire Solr

Extension application from parameters in the `solrserver-config.xml` file. In the base configuration, the `port` parameter specifies the standard Solr port number, 8983. If you set up the Guidewire Solr Extension with a different port number, modify the `port` parameter to match your configuration.

In the base configuration, the `host` parameter specifies `localhost`, which generally is correct for development environments. For production environments however, Guidewire requires that you set up the application server instance for the Guidewire Solr Extension on a host separate from the one that hosts ClaimCenter. For production environments, you must modify the `host` parameter to specify the remote host where the Guidewire Solr Extension runs.

If you configure free-text search for embedded operation, the plugins connect to the Guidewire Solr Extension without using the HTTP protocol. With embedded operation, the Guidewire Solr Extension runs as part of the ClaimCenter application, not as an external application. With embedded operation, the plugins ignore any host name and port number parameters specified in `solrserver-config.xml`. Free-text search does not support embedded operation in production environments.

Enabling and disabling the free-text plugins

The free-text plugins are disabled in the base configuration of ClaimCenter. After you enable the free-text plugins, you must perform the following actions for the plugin implementations to fully operate:

- Set the `FreeTextSearchEnabled` parameter in `config.xml` to `true`.
- Enable the `CCSolrMessageTransport` message destination, which the `ISolrMessageTransportPlugin` requires.

After you enable the free-text plugins, use the `FreeTextSearchEnabled` parameter to toggle them on and off, along with other free-text resources.

Running the free-text plugins in debug mode

You can run the free-text plugins in debug mode. With debug mode enabled, the plugins generate messages on the server console to help you debug changes to free-text search fields. The free-text plugin implementations have a `debug` plugin parameter that lets you enable and disable debug mode. You can enable debug separately for each plugin.

In the base configuration, the `debug` parameters are set to `true`. To use free-text search in a production environment, set the `debug` parameters for each plugin to `false`.

IMPORTANT: You must set the `debug` parameters to `false` in a production environment.

Free-text load and index plugin and message transport

ClaimCenter provides the free-text load and index message transport to send changed claim contact data to the Guidewire Solr Extension for loading and incremental indexing. In the base configuration of ClaimCenter, the plugin is disabled.

The primary components of this message transport are:

- A `MessageTransport` plugin implementation with the following qualities:
 - The implementation class is `gw.solr.CCSolrMessageTransportPlugin`.
 - The plugin name in the Plugins Registry in Studio is `SolrMessageTransportPlugin`
 - This class implements the interface called `ISolrMessageTransportPlugin`, which is a subinterface of `MessageTransport` with additional methods. In the base configuration, in the Plugins Registry, this plugin implementation specifies its interface name as `ISolrMessageTransportPlugin` instead of `MessageTransport`.
- A messaging destination

Generally, you do not need to modify the `CCSolrMessageTransportPlugin` implementation if you add or remove free-text search fields.

To edit the Plugins Registry item for the `CCSolrMessageTransportPlugin` interface, in the **Project** window in Studio, navigate to **configuration > config > Plugins > registry**, and then open `ISolrMessageTransportPlugin.gwp`.

See also

- “Messaging and events” on page 339

Message destination for free-text search

The implementation of `CCSolrMessageTransportPlugin` depends on the `CCSolrMessageTransport` message destination. If the message destination is not enabled or is enabled but not started, the plugin cannot send index documents to the Guidewire Solr Extension. Use the **Messaging** editor in Studio to enable the `PCSolrMessageTransport` destination. Use the **Event Messages** page on the **Administration** tab in the application to start and stop the destination.

If you enabled the free-text search feature, enable the message transport implementation. The implementation class is `gw.solr.CCSolrMessageTransportPlugin`, which is an implementation of the `MessageTransport` plugin interface.

The plugin is enabled in the Plugins Registry in the base configuration. However, you must enable the relevant messaging destination in the **Messaging** editor in Studio.

Enable the message destination for free-text search

Procedure

1. In the **Project** window in Studio, navigate to **configuration > config > Messaging**, and then open `messaging-config.xml`.
2. In the list on the left, select the row for message ID 69.
3. Ensure that the following fields are set to the following values.

Field	Value
Enabled	The check box is selected.
Destination ID	69
Name	<code>Java.MessageDestination.SolrMessageTransport.Policy.Name</code>
Transport plugin	<code>ISolrMessageTransportPlugin</code> This value is not the fully-qualified name. This value is the name for the plugin in the Plugins Registry editor.
Poll interval	1000
Events	<ul style="list-style-type: none">• <code>ClaimExceedsLargeLoss</code>• <code>ClaimResync</code>
Chunk Size	100000
Poll Interval	1000
Max Retries	3
Initial Retry Interval	1000
Message Without Primary	
The Single Threaded option is selected.	

ISolrMessageTransportPlugin plugin parameters

The Plugins Registry item with name `ISolrMessageTransportPlugin` has the following plugin parameters.

Plugin parameter	Description	Default value
commitImmediately	Whether the Guidewire Solr Extension indexes and commits each new or changed index document before receiving and indexing the next one. If you set this parameter to <code>false</code> , the Guidewire Solr Extension receives a batch of index documents from ClaimCenter before it indexes and commits them. You configure the batch size in the <code>autocommit</code> section of the <code>solrconfig.xml</code> file.	<code>false</code>
debug	For development servers only, specifies whether to generate messages on the server console and in the server log to help debug changes to free-text search fields. For production, always set to <code>false</code> .	<code>true</code>

Free-text search plugin

ClaimCenter provides the free-text search plugin `ISolrSearchPlugin` to send free-text search requests to the Guidewire Solr Extension and receive the search results. In the base configuration of ClaimCenter, the plugin is disabled. The Plugins Registry specifies the following Gosu implementation:

```
gw.solr.CCSolrSearchPlugin
```

If you add or remove free-text search fields from your configuration, you must modify the implementation of `CCSolrSearchPlugin`.

To edit the Plugins Registry item for the `ISolrSearchPlugin` interface, in the **Project** window in Studio, navigate to **configuration > config > Plugins > registry**, and then open `ISolrSearchPlugin.gwp`.

See also

- *Configuration Guide*

ISolrSearchPlugin plugin parameters

The Plugins Registry item for `ISolrSearchPlugin` has the following plugin parameters.

Plugin parameter	Description	Default value
debug	Whether to generate messages on the server console and in the server log to help debug changes to free-text search fields. For a production server, always set to <code>false</code> .	<code>true</code>
chunkSize	Determines the number of query results that the Guidewire Solr Extension sends to PolicyCenter at one time. The Guidewire Solr Extension sends these blocks of data, or chunks, until all the query results for a search request have been sent.	25
pageIteratorCacheTimeout	Determines in seconds how long ClaimCenter holds a chunk, or page, of results from the Guidewire Solr Extension before discarding it and requesting fresh results.	300

Management integration

The management integration API enables external applications to trigger ClaimCenter actions or retrieve ClaimCenter data. For example, a console application can call the API to trigger ClaimCenter metrics to be published to an external location.

Examples of retrieved data include lists of the application's current users, its running batch processes, and its active database connections. The API can also modify some ClaimCenter configuration parameters. The following types of ClaimCenter data can be retrieved.

Configuration parameter settings

Retrieve configuration parameter settings and modify the value of certain parameters. Changed parameter settings take effect on the server immediately, without requiring the server to restart. Modified parameter settings do not persist between application sessions. After the server shuts down, ClaimCenter discards the dynamic changes.

When the application server is restarted, the parameters originally specified in the `config.xml` configuration file are used.

Batch processes

Retrieve the list of currently running batch processes.

Users

Retrieve the list of current users and user sessions.

Database connections

Retrieve lists of the active and idle database connections.

Notifications

Retrieve the list of notifications pertaining to users locked out due to excessive login failures.

Communication between an external application and the API can occur by using a standard protocol, such as JMX (Java Management Extensions) or SNMP (Simple Network Management Protocol).

Within ClaimCenter, the management integration API is implemented in the form of a plugin. The plugin initializes the desired communication channel and processes API calls that arrive through the channel. Multiple implementations of the management integration API can be registered in ClaimCenter to communicate with multiple external applications. Each API implementation must have a unique name.

ClaimCenter exposes data through both its user interface and the management integration API.

In the **ClaimCenter Server Tools** tab, application data is retrieved and listed by selecting the various Sidebar menu items. The **Server Tools** tab is accessible to users who have the `soapadmin` permission.

The management integration API is defined by the `ManagementPlugin` interface. A ClaimCenter plugin class can be defined to implement the interface and handle API calls made by external applications.

Management plugin examples

The source code for two example management integration API plugin classes is provided.

- The `JMXManagementPlugin` class returns data requested by an external application. Communication between the plugin API and the external application uses the JMX protocol.
- The `AWSManagementPlugin` does not return data, but instead performs a repeated action. The plugin publishes ClaimCenter metrics to AWS (Amazon Web Services) Cloudwatch. The publishing operation is scheduled to repeat intermittently. External applications can access the published metrics as desired.

Java source code for the example plugins can be found in the `java-examples.zip` file, which is located in the `ClaimCenter` directory.

Extract the files in the archive file. The source files for the example plugins will be located in the following `ClaimCenter` directory.

```
examples/src/examples/p1/plugins/management
```

Initialization methods

The `ManagementPlugin` interface provides methods to start and stop the communication channel used to receive and respond to API calls.

```
function start()  
function stop()
```

The `JMXManagementPlugin` example plugin implements these methods to set up and close the JMX communication channel it uses to receive and respond to API requests. The `AWSManagementPlugin` `start` method gains access to AWS Cloudwatch and schedules a daemon thread to intermittently perform the metric publishing operation. Its `stop` method kills the thread.

Methods to register and unregister management beans

The `ManagementPlugin` interface also defines methods that can set up a repository of the data resources that the plugin can return.

```
function registerBean(bean : gw.plugin.management.GWMBean)  
function unregisterBean(bean : gw.plugin.management.GWMBean)  
function unregisterBeanByNamePrefix(prefix : String)
```

The `GWMBean` arguments reference Guidewire management beans which are ClaimCenter resources, such as current users, running batch processes, and configuration parameters. Each `GWMBean` contains information about itself, including its name, description, attribute values, notifications, and operations it can perform.

The `JMXManagementPlugin` methods register and unregister the received bean argument. The `AWSManagementPlugin` does not need a data repository, so its example methods log notification messages when they are called.

Notification method

The `ManagementPlugin` interface defines a method to have a particular `GWMBean` generate a notification. A notification can signal that its state has changed or an event or problem has occurred.

```
function sendNotification(notification : Notification)
```

The `notification` argument references a `Notification` object containing information, including the name of the relevant `GWMBean`. The Gosu `Notification` class is similar to the `javax.management.Notification` class. The plugin can utilize the Java `Notification` framework if desired, but it is not mandatory.

The `JMXManagementPlugin` `sendNotification` method demonstrates how to utilize the Java Notification framework to have the relevant `GWMBean` send a notification. If the referenced `GWMBean` exists, the properties of the method's notification argument are passed to the `javax.management.Notification` constructor. The `AWSManagementPlugin` does not support notifications and so its implementation of the `sendNotification` method is empty.

Authorization callback method

The `ManagementPlugin` interface defines a method to receive a `ManagementAuthorizationCallbackHandler` object. The referenced callback handler implements a method called `hasManagementPermission`. The handler's `hasManagementPermission` method is called to authenticate whether the current user has the required management permission to perform the requested operation.

```
function setAuthorizationCallbackHandler(handler : ManagementAuthorizationCallbackHandler)
```

The `JMXManagementPlugin` implementation saves the handler argument in a local variable and passes it to the `JMXRMIConector` constructor when the JMX communication channel is created. The callback method is ultimately referenced by the `JMXAuthenticatorImpl` class which calls its `hasManagementPermission` method to authenticate the user. For details concerning the `JMXAuthenticator` class and related management classes and methods, refer to the `javax.management.remote` documentation.

Calling the example `JMXManagementPlugin`

To demonstrate how an external application might call the example `JMXManagementPlugin`, perform the operations described in the following sections.

Prepare the `JMXManagementPlugin` implementation

Use IntelliJ with OSGi Editor to prepare the `JMXManagementPlugin` implementation for deployment to Guidewire Studio.

Before you begin

- Extract the example plugins from the `java-examples.zip` file, which is located in the `ClaimCenter` installation directory.
- “Set up a project with an OSGi plugin module” on page 56

About this task

You use the example implementation of the `JMXManagementPlugin` interface and IntelliJ with OSGi Editor to create a JAR file that Guidewire Studio can use.

Procedure

1. If IntelliJ with OSGi Editor is not open, type the following command.

```
gwb pluginStudio
```

2. Create a package for the management plugin files by right-clicking `src` and choosing **New > Package**.
3. In **New Package**, type `examples.pl.plugins.management` and then press **Enter**.
4. Copy the contents of the `examples/src/examples/pl/plugins/management` directory to the `src/examples/pl/plugins/management` directory that you just created in your project.

This operation places the example plugin source files into the IntelliJ with OSGi Editor **Project** structure.

What to do next

- “Compile and install your OSGi plugin as an OSGi bundle” on page 60

Register the JMXManagementPlugin implementation

You must register the Java implementation of `JMXManagementPlugin` before you can call the plugin from an external application.

Before you begin

- “Prepare the `JMXManagementPlugin` implementation” on page 279

Procedure

- In the Studio Project window, select **configuration > config > Plugins > registry**.
- Right-click the **registry** directory and select **New > Plugin**.
The **Plugin** dialog appears.
- Enter a plugin **Name** of `JMXManagementPlugin`. Enter the **Interface** name of `ManagementPlugin`.
- Click **OK**.
Studio creates the plugin and shows its initial settings in the plugin registry.
- In the Plugins registry main pane for the new plugin, click the plus sign (+) and select **Add OSGi Plugin**.
- In the **Service PID** field, type the fully qualified Java class name for your OSGi implementation class.
`examples.pl.plugins.management.JMXManagementPlugin`

What to do next

You can call the `JMXManagementPlugin` from an external application. See “Calling the `JMXManagementPlugin` from an external application” on page 280.

Calling the JMXManagementPlugin from an external application

The following Java code demonstrates how an external application can retrieve a ClaimCenter system attribute called `HolidayList` by calling the example `JMXManagementPlugin`.

```
package com.mycompany.xx.integration.jmx;

import java.util.HashMap;
import java.util.Map;

import javax.management.MBeanServerConnection;
import javax.management.ObjectName;
import javax.management.remote.JMXConnector;
import javax.management.remote.JMXConnectorFactory;
import javax.management.remote.JMXServiceURL;
import javax.naming.Context;

public class TestJMXClientConnector {

    public static void main(String[] args) {
        try {
            // *** Start an RMI connector for the JMX server
            // Note: An RMI connector is optional. If not desired, specify a negative RMI port
            // number in the RMI URL so the connector will not be started.

            // The syntax of the RMI URL is either of the following:
            // "service:jmx:rmi://[host]:[rmiPort]/jndi/rmi://[host]:[rmiPort]/jrmp"
            // "service:jmx:rmi:///jndi/rmi://[host]:[rmiPort]/jrmp"

            // If rmiPort is not specified, a default value of 1099 is used.
            // If rmiPort is set to a negative value, the RMI connector will not be started.
            // Remote monitoring can still be configured using "com.sun.management.jmxremote.port"
            // or "com.sun.management.jmxremote.authenticate" with other JVM-supported properties.

            // Note: The RMI connector does not work on JBoss 6.1.1. For JBoss 6.1.1, it is
            // recommended that the rmiPort be set to a negative number so the connector will not
            // be started.

            // Specify the URL address of the RMI connector server
            JMXServiceURL address = new JMXServiceURL(
                "service:jmx:rmi:///jndi/rmi://akitio:1099/jrmp");
        }
    }
}
```

```
// The creation environment map
Map creationEnvironment = null;

// Create the JMXConnectorServer
JMXConnector cntor = JMXConnectorFactory.newJMXConnector(address, creationEnvironment);

// The environment may contain the user's credentials and desired other information
Map environment = new HashMap();
environment.put(Context.INITIAL_CONTEXT_FACTORY,
    "com.sun.jndi.rmi.registry.RegistryContextFactory");
environment.put(Context.PROVIDER_URL, "rmi://localhost:1099");
String[] credentials = new String[]{"su", "cc"};
environment.put(JMXConnector.CREDENTIALS, credentials);

// Connect the environment to the RMI connector
cntor.connect(environment);

// *** Connect with the remote MBeanServer
// This exposes Guidewire management beans via JMX.

// Obtain a stub to the remote MBeanServer
MBeanServerConnection mbsc = cntor.getMBeanServerConnection();

// Call the remote MBeanServer
String domain = mbsc.getDefaultDomain();
ObjectName delegate =
    ObjectName.getInstance("com.guidewire.pl.system.configuration:type=configuration");

// Retrieve the "HolidayList" attribute
String holidayList = (String)mbsc.getAttribute(delegate, "HolidayList");
System.out.println(holidayList);
}

catch (Exception e) {
    throw new DisplayableException(e.Message);
}
}
}
```


Other plugin interfaces

Plugins are software modules that ClaimCenter calls to perform an action or calculate a result. This topic describes plugin interfaces that are not discussed in detail elsewhere in this documentation.

SharedBundlePlugin marker interface

When a plugin is called, the default ClaimCenter behavior is to create a new bundle. This new bundle is separate from any bundle that exists at the time the plugin is called. The contents of the existing bundle are copied to the new bundle. The new bundle is then designated to be the current bundle. Plugin operations that effect the bundle, such as adding, removing, and changing objects in the bundle, are applied to the current bundle.

After the plugin has finished its operations, the changes performed on the bundle are not automatically applied to the original bundle for subsequent commitment. For the contents of the plugin bundle to be committed, the caller of the plugin must combine the returned plugin bundle with the original bundle.

In situations where it is desirable for the plugin to directly modify the original bundle, the plugin class can implement the SharedBundlePlugin interface. The SharedBundlePlugin interface acts as a marker that directs ClaimCenter not to create a new bundle when the plugin is called. The original bundle remains the current bundle so that changes performed by the plugin affect the original bundle directly. There is no need for the plugin caller to combine the returned plugin bundle with the original bundle.

A plugin class that implements the SharedBundlePlugin interface must specify the @Export annotation as part of the class definition.

Credentials plugin

The CredentialsPlugin plugin provides a secure way to retrieve the user name and password for user authentication to gain access to an external system.

This plugin is located at **configuration > config > Plugins > registry > CredentialsPlugin**.

The CredentialsPlugin is called by the static getCredentialsFromPlugin method in the CredentialsUtil class.

```
getCredentialsFromPlugin(key : String) : UsernamePasswordPairBase
```

The key argument references the relevant user name and password credentials.

The plugin provides a retrieveUserNameAndPassword method.

```
retrieveUserNameAndPassword(key : String) : UsernamePasswordPairBase
```

The method returns a `UsernamePasswordPairBase` object that contains the retrieved user's name in its `Username` field and the password in the `Password` field.

Base configuration implementation

The base configuration of ClaimCenter provides an implementation of the `CredentialsPlugin` plugin in the `Gosu gw.plugin.credentials.impl.CredentialsPlugin` class. The implementation retrieves the user name and password from a file that can optionally be encrypted. In a production environment, best practice is to write an implementation that uses an ActiveDirectory or LDAP resource to retrieve credentials.

This implementation recognizes the following plugin parameters. You can set values for those parameters in the Studio plugin registry editor.

`credentialsFile`

Name of the file containing the user name and password

`FileEncryptionId`

String identifier of the `Encryption` plugin to use to decrypt the `credentialsFile`

`PasswordEncryptionId`

String identifier of the `Encryption` plugin to use to decrypt the password

The `retrieveUsernameAndPassword` method opens and parses a credentials file to extract and return the user's name and password. If the key is not found or the key does not reference existing credentials, the method returns `null`.

If the credentials file specified in the plugin's `credentialsFile` parameter is encrypted, it is decrypted by calling the `Encryption` plugin referenced by the `FileEncryptionId` plugin parameter of the `CredentialsPlugin`. The password can also be encrypted, in which case the decrypting `Encryption` plugin is referenced by the `PasswordEncryptionId` parameter.

By defining multiple environments for the plugin in the Studio plugin registry editor, you can use a different `credentialsFile` in each environment. For example, the plugin can define a development environment and a `credentialsFile` parameter that reads user names and passwords from a locally stored XML file. The plugin can also define a production environment where the `credentialsFile` references a different file, such as an encrypted file stored in an external repository in a central JNDI (Java Naming and Directory Interface) registry.

The structure of a plain-text XML credentials file is defined in the `Credentials.xsd` schema file. A sample XML file that conforms to the schema is shown below. Note that the example user names and passwords are stored in plain text. Such a storage mechanism is not secure and is not recommended for a production environment.

```
<?xml version="1.0"?>
<tns:CredentialsArray xmlns:tns="http://guidewire.com/credentials">
  <tns:CredentialsElem key="AcmeWebService">
    <username>John Doe</username>
    <password>JohnsSecurePassword</password>
  </tns:CredentialsElem>
  <tns:CredentialsElem key="AcmeFinanceIntegration">
    <username>Jane Doe</username>
    <password>JanesSecurePassword</password>
  </tns:CredentialsElem>
</tns:CredentialsArray>
```

The base configuration includes an implementation of the `Encryption` plugin that demonstrates techniques for processing encrypted credentials files. The `Gosu PBEEncryptionPlugin` class implements the `IEncryption` interface and demonstrates the processing of a credentials file encrypted by the group of `javax.crypto.spec.PBE*` classes.

Preupdate handler plugin

By default, ClaimCenter calls the `IPreUpdateHandler` plugin whenever it commits a bundle to the database.

ClaimCenter executes the plugin first before applying any existing preupdate rule set, thereby enabling preprocessing of the objects in the bundle.

Preprocessing objects in a bundle

The following list describes scenarios in which it is beneficial to preprocess the objects in a bundle before applying the rules in a preupdate rule set.

Scenario	Notes
If there is a need to apply the preupdate rules to one of the following:	A rule that adds or changes a bundle object does not trigger another running of the rule set. Therefore, any objects that a rule adds or modifies is not processed by the rule set. If an added or changed object must be processed by the preupdate rule set, the add or change operation must be performed before the rule set is run. The <code>IPreUpdateHandler</code> plugin is an appropriate place to perform such operations. <ul style="list-style-type: none"> • To an object added to the bundle • To a modified object in the bundle
If there is a need to remove objects from the bundle	Only the addition or modification of an entity in the bundle triggers a preupdate rule set (if one exists for that particular entity type). Removing an object does not trigger the preupdate rule set. Because of this behavior, plugin code is the preferred location for preprocessing removed objects. As an example, if ClaimCenter deletes a check, it also deletes its payment objects and related subobjects. The action results in retiring <code>Transaction</code> entities, but ClaimCenter does not add an entity to the bundle or modify an existing bundle entity. Therefore, this action does not trigger the <code>TransactionSet</code> preupdate rules. In addition, this behavior can indirectly affect related operations. Suppose that a property on a related entity type tracks the sum of multiple <code>Payment</code> objects. To maintain the total payment amount requires tracking all new, modified, and deleted <code>Payment</code> objects. However, a typical preupdate rule fails to catch deleted <code>Payment</code> objects. In such cases, it is useful for the <code>IPreUpdateHandler</code> plugin to track and perform the desired operation.
If there is a need to process the objects in the bundle in a particular order	The plugin code can be written to process the bundle's objects in any desired order.

The `executePreUpdate` method

The `IPreUpdateHandler` plugin must implement the `executePreUpdate` method.

```
override function executePreUpdate(context : PreUpdateContext)
```

In Gosu code, the `PreUpdateContext` object includes properties that contain lists of added, modified, and deleted objects in the current bundle.

InsertedBeans

An unordered list of added objects

UpdatedBeans

An unordered list of modified objects

RemovedBeans

An unordered list of deleted objects

In Java code, the argument's list properties can be retrieved by calling the methods `getInsertedBeans`, `getUpdatedBeans`, and `getRemovedBeans`.

The `executePreUpdate` method has no return value.

The `executePreUpdateRules` method

It is possible to apply the existing preupdate rule set to a single object by passing the object to the `executePreUpdateRules` method of the `PreUpdateUtil` class, defined in the `gw.api.preupdate` package:

```
function executePreUpdateRules(bean : Object)
```

The method accepts a bean argument that references the object to be processed by the existing preupdate rule set. The object must be a valid instance of an entity. If no preupdate rule exists for the object, the object remains unchanged.

It is possible to call the `executePreUpdateRules` method at any time. Its use is not restricted to the `IPreUpdateHandler` plugin.

The method has no return value.

Base configuration implementation

ClaimCenter provides a default implementation of the `IPreUpdateHandler` plugin in Gosu class `CCPreupdateHandlerImpl`, in the `gw.plugin.preupdate.impl` package. Guidewire registers the plugin class in the Studio plugin registry under the name `IPreUpdateHandler`. Guidewire enables this plugin by default in the base configuration.

The `CCPreupdateHandlerImpl` class extends abstract Gosu class `CCPreupdateAbstractHandlerImpl`. The `executePreUpdate` method of the child class calls the abstract parent's superclass version of the method. The superclass method processes Workers Compensation exposures and Reinsurance plugin information. Guidewire requires that every bundle commit execute the plugin superclass operations. Thus, any custom plugin code that you write must either retain the child method's call to the superclass or perform the superclass operations itself in the same order as the superclass.

The `CCPreupdateHandlerImpl` class implements the `SharedBundlePlugin` marker interface. The `SharedBundlePlugin` interface affects which bundle the plugin modifies whenever the plugin adds, changes, or removes an object from the bundle. The standard plugin behavior is to create a second bundle during the plugin call. The plugin then performs the bundle modifications on this second bundle. However, a plugin that implements `SharedBundlePlugin` does not create a second bundle. Thus, ClaimCenter applies the modifications to the current bundle.

See also

- “`SharedBundlePlugin` marker interface” on page 283

General recommendations

1. Guidewire recommends that you keep related preupdate logic together in the same location, either in the Preupdate rules, or in Gosu code that calls the `IPreupdateHandler` plugin. For example, if preupdate rules already exist for the entity for which you want to provide additional preupdate logic, continue to use the Preupdate rules to handle that entity. Conversely, if the `IPreupdateHandler` plugin code already handles preupdate logic for an entity, continue to use the plugin implementation to handle preupdate operations for that entity. As you make design decisions, be mindful of any maintenance or upgrade issues with your choice of preupdate handling.
2. Guidewire recommends that you perform load and performance testing of the new rules if you choose to create additional Predate rules. In general, rules that use an incorrect rule hierarchy and rule order can introduce performance issues. Thus, perform load and performance testing to verify that the new rules are working efficiently.

Validation plugin

The Validation plugin enables validation operations to be performed on a `Validatable` object. The plugin is typically called to run a validation rule set.

The `gw.plugin.validation.IValidationPlugin` interface defines the following single method for the plugin.

```
validate(bean : Validatable)
```

The `bean` argument specifies the object to validate. The method has no return value.

The base configuration implements the Validation plugin in the `CCValidationPluginImpl` class. The plugin is registered in the Plugins registry under the name `IValidationPlugin`. The plugin `validate` method runs a validation rule set.

Work item priority plugin

Configure how ClaimCenter calculates work item priority for workflow steps by implementing the `IWorkItemPriorityPlugin` Work Item Priority plugin. The interface has a single method called `getWorkItemPriority` which takes a `WorkflowWorkItem` parameter. Your method implementation returns a priority as a non-negative integer. A higher priority integer indicates workflow steps to process before other workflow steps with lower priorities. If there is no plugin implementation, the default workflow step priority is zero.

Prioritization affects only work items of type `WorkflowWorkItem` or its derivatives.

Exception and escalation plugins

There are several optional exception and escalation plugins. By default, they just call the associated rule sets and perform no other function. Implement your own version of the plugin and register it in Guidewire Studio™ if you want something other than the default behavior.

Use the plugins for the following tasks.

- Add additional logic before or after calling the rule set definitions in Studio.
- Completely replace the logic of the rule set definitions in Studio.

One reason you might want to completely replace the logic of the rule set definitions in Studio is to make your code more easily tested using unit tests.

The following table lists the exception and escalation plugins.

Name	Plugin interface	Default action
Activity escalation	<code>IActivityEscalationPlugin</code>	Calls the activity escalation rule set
Group exceptions	<code>IGroupExceptionPlugin</code>	Calls the group exception rule set
User exceptions	<code>IUserExceptionPlugin</code>	Calls the user exception rule set

Sending emails

To send emails, define an email message destination and configure the `emailMessageTransport` plugin.

Defining the email message destination

To send email messages from ClaimCenter, define an email message destination to process them.

The base configuration defines an example email message destination. To view the destination in Studio, navigate to the following location and open the `messaging-config.xml` file.

This file is located at `configuration > config > Messaging`.

The example destination is assigned a unique ID value of 65, which references an external system defined in the Studio **Messaging** editor. The ID for the email message destination cannot be changed, but other parameters associated with the destination can be modified.

Configuring the email message transport plugin

To send email messages from ClaimCenter, configure the email message transport implemented in the `emailMessageTransport` plugin.

This plugin is located at `configuration > config > Plugins > registry > emailMessageTransport`.

In the base configuration of ClaimCenter, the email transport that is registered is the sample class `EmailMessageTransport`. This sample class is for demonstration purposes only and cannot be used in a production environment. Instead, you can use the `JavaxEmailMessageTransport` class in the `gw.plugin.email.impl` package.

This plugin class implements the `gw.plugin.messaging.MessageTransport` interface, which is used to send an email. Alternatively, create your own email message transport class that implements the same interface.

`JavaxEmailMessageTransport` class

The `JavaxEmailMessageTransport` class can be used as the basis for an email messaging transport in a production environment. The class is located in the `gw.plugin.email.impl` package.

The `JavaxEmailMessageTransport` class supports the following parameters that can be specified in the registry.

CredentialsPlugin.Key

Optional. String key value stored in the `Credentials` plugin. If this parameter is specified and the `Credentials` plugin is enabled, the user name and password values to use in an SSL authenticated login are retrieved from the plugin. The plugin's properties can be encrypted, which enables secure user and password strings.

Default: `EmailMessageTransport`

Debug

Optional. Boolean value to enable debugging output from the email host.

Default: `false`

defaultSenderAddress

Default *From* email address to use for outbound email. For example, `jdoe@mymail.com`.

The `defaultSenderAddress` and `defaultSenderName` parameter values are used if both the `Sender` and `Sender.EmailAddress` properties are not specified in the `Email` object.

defaultSenderName

Default sender's name to use for outbound email. For example, John Doe.

*mail.**

Optional. Specifies various `mail.*` properties.

Properties recognized in the base configuration are listed below. Additional `mail.*` properties can be recognized by writing configuration code.

- `mail.transport.protocol`: Specifies the desired protocol. Default: "smtp"
- `mail.smtp.host`: Specifies the SMTP email application that sends emails.
- `mail.smtp.port`: Specifies the SMTP port number.

If any `mail.*` parameters are specified, the values specified in the `smtpHost` and `smtpPort` parameters are ignored.

If no `mail.*` parameters are specified and a password is specified, either in the `Password` parameter or retrieved from the `Credentials` plugin, the following properties and values are defined.

- `mail.smtp.ssl.enable`: Set to `true`.
- `mail.smtp.auth`: Set to `true`.

To support the TLS security standard, set the following `mail.*` parameters:

`mail.smtp.auth`

Set to `true`.

`mail.smtp.starttls.enable`

Set to `true`.

`mail.smtp.host`

The SMTP email application that sends emails

mail.smtp.port

The SMTP port number

Also set the `Username`, `Password`, `defaultSenderName`, and `defaultSenderAddress` to appropriate values for your TLS requirements.

password

Optional. Login password.

The `Password` parameter is plain text and not secure. It is recommended to specify user name and password values in encrypted properties in the `Credentials` plugin. See the `CredentialsPlugin.Key` parameter.

Default: Not specified

smtpHost

Name of the SMTP email application that sends emails.

If any `mail.*` parameters are specified, the `smtpHost` parameter is ignored.

smtpPort

SMTP email port.

If any `mail.*` parameters are specified, the `smtpPort` parameter is ignored.

Default: 25 for non-authenticated logins; 465 for authenticated logins. See the `Password` parameter.

useDefaultAsSender

Optional. Boolean value indicating whether to use the `defaultSenderName` and `defaultSenderAddress` parameters to identify the email sender. If `true`, the parameter values are used even if the email explicitly specifies a sender and address.

Default: `false`

useMessageCreatorAsUser

Optional. Boolean value to specify on whose behalf to retrieve a document attached to the email. Possible personas are either the user who generated the email (`true`) or the system user (`false`).

Default: `false`

Username

Optional. Login user name.

The `Username` parameter is plain text and not secure. It is recommended to specify user name and password values in encrypted properties in the `Credentials` plugin. See the `CredentialsPlugin.Key` parameter.

Default: Not specified

In the base configuration, starting the plugin initializes the environment by performing the following tasks:

- Reads the default values of the plugin parameters.
- Sets the SMTP host name and port.
- If a password is configured in the `emailMessageTransport` plugin, enables an SSL authenticated login.

The class implements the following important method:

send(message : entity.Message, transformedPayload : String)

The method accepts the following arguments.

message

Message entity instance to send

transformedPayLoad

The transformed payload of the message

The method does not return a value.

In the base configuration, the `send` method performs the following operations.

- If either the `email.Sender` or `email.EmailAddress` properties are not set, uses the `defaultSenderName` and `defaultSenderAddress` values configured in the `emailMessageTransport` plugin.
- Retrieves a `Session` object. If a `Session` does not already exist, one is created. If an authenticated login is performed, access to the `Session` object is restricted to the authenticated user. Also, if the `emailMessageTransport` plugin's `debug` parameter is `true`, debugging on the `Session` is enabled.
- Creates and initializes a new `HtmlEmail` object.
- If the SMTP host name is not an empty string, the email is sent.

If an error occurs that causes a `MessagingException` exception, the method catches that exception. If the exception is caused by an invalid email address, the invalid address is removed from the recipient list and an attempt is made to resend the email to any remaining recipients. The removal event is written to the log, but no further action is taken to recover from an invalid address. To modify this behavior, such as to create an activity or forward the removed email to a postmaster for additional processing, edit the `handleMessageException` method in the `JavaxEmailMessageTransport` class.

Other exceptions set the error description on the message and report the error.

IEmailTemplateSource plugin

The `IEmailTemplateSource` plugin template enables ClaimCenter to retrieve one or more email templates. To see the plugin registry, navigate in the following location in the Studio **Project** window:

configuration > config > Plugins > registry > IEmailTemplateSource.gwp

In the base configuration, Guidewire provides the following demonstration plugin implementation, registered in `IEmailTemplateSource.gwp`:

```
gw.plugin.email.impl.LocalEmailTemplateSource
```

The `LocalEmailTemplateSource` class constructs an email template from files located in the Studio **resources** folder:

configuration > config > resources > emailtemplates

Method `getEmailTemplates` on the implementation class searches for an email template.

```
getEmailTemplates(locale : ILocale, valuesToMatch : Map) : IEmailTemplateDescriptor[]
```

This method accepts the following arguments.

Parameter	Description
<code>locale</code>	Locale to search for. The argument can be <code>null</code> . If <code>locale</code> is not <code>null</code> , the search is performed in the resource > emailtemplates folder for a subdirectory whose name matches the specified <code>locale</code> value. A match is tested in the following order. <ul style="list-style-type: none"> • Language + Country + Variant • Language + Country • Language • Default language as specified in the <code>config.xml</code> configuration parameter <code>DefaultApplicationLanguage</code>.
<code>valuesToMatch</code>	Keys to match include <code>topic</code> , <code>name</code> , <code>keywords</code> , and <code>availablesymbols</code> . The <code>availablesymbols</code> key is matched against the template's <code>requiredsymbols</code> .

The method returns an array of zero or more `IEmailTemplateDescriptor` objects that match the locale and specified values to match. If no matches are found, the returned array is empty.

Defining base URLs for fully qualified domain names

If ClaimCenter generates HTML pages, it typically generates a base URL for the HTML page using a tag such as `<base href="...">` at the top of the page. In almost all cases, the default implementation in ClaimCenter generates the most appropriate base URL, based on settings in `config.xml`. This implementation returns a base URL as `scheme://host:port` that ignores values in the HTTP header. The `:port` portion of the URL is omitted for the following cases:

- The scheme is `http` and the port is 80.
- The scheme is `https` and the port is 443.

In some cases, this behavior is inappropriate. For example, suppose you deploy ClaimCenter behind a load balancing router that handles Secure Socket Layer (SSL) communication. In such a case, the external URL would include the prefix `https://`. The load balancer handles security and forwards a non-secure HTTP request to ClaimCenter with a URL prefix `http://`. The default implementation of the base URL includes the URL prefix `http://`.

The load balancer would not typically parse the HTML enough to know about this problem, so the base URL starts with `http` instead of `https`. Image loading and display then fail because the browser tries to load the images relative to the `http` URL. The load balancer rejects the insecure requests because they do not use HTTPS/SSL.

Avoid this problem by writing a base URL builder plugin and registering it with the system. The plugin class implements the `gw.plugin.baseurlbuilder.IBaseURLBuilder` interface.

To handle a deployment that uses a load balancer, the base URL builder plugin can examine the HTTP request's header. You can designate a plugin parameter to indicate that the request came from the load balancer, and return a base URL with the prefix `https` instead of `http`.

To enable the custom plugin, register it with ClaimCenter. In the Guidewire Studio **Project** window, navigate to **configuration > config > plugin > registry**. Right-click **registry** and select **New > Plugin**.

The base URL builder plugin must implement the `IBaseURLBuilder` methods described in the following sections.

Method: `getApplicationBaseURL`

```
String getApplicationBaseURL(HttpServletRequest request)
```

The `request` argument must specify the desired scheme, server port, server name, and context path for the web application. The scheme is a `String`, typically `http` or `https`. If the scheme is not specified, `http` is used by default. The server name can be either a name or IP address of the server. The port can be omitted if either of the following conditions exists.

- The scheme is `http` and the port is 80.
- The scheme is `https` and the port is 443.

The method uses the `request` argument fields to construct and return a `String` containing the base URL of the web application, such as `http://servername:8080/webapp`.

Method: `getPageBaseURL`

```
String getPageBaseURL(HttpServletRequest request)
```

The `request` argument must specify the desired scheme, server port, server name, and request URI for the HTML page. The scheme, server port, and server name fields are processed in the same manner as in the `getApplicationBaseURL` method.

The method uses the `request` argument fields to construct and return a `String` containing the base URI of the relevant HTML page, such as `http://servername:8080/webapp/path/SomePage.jsp`.

Implementations of `IBaseURLBuilder`

ClaimCenter provides the following two implementations in the `gw.plugin.baseurlbuilder` package.

You can use either of these implementations as is. Alternatively, copy one of these classes and modify the code to suit your business needs. ClaimCenter provides the source code for these implementations in a JAR file, not in the configuration/gsrc hierarchy. In Studio, use **Navigate > Class** to locate and open the Gosu source files. The source code of the default implementation is not available.

FixedBaseUrlBuilder

This implementation is the same as the default implementation but supports forcing the base URL to a value that you provide. To avoid cross-site scripting (XSS) vulnerabilities, any client-modifiable data values, including those in the HTTP Host header, are ignored.

The plugin parameter `FqdnForUrlRewrite` is available for you to force the value of the base URL. This parameter is not set by default. If you enable browser-side integration features, you must specify this parameter to rewrite the URL for the external fully qualified domain name (FQDN). The JavaScript security model prevents access across different domains. Therefore, if ClaimCenter and other third-party applications are installed on different hosts, the URLs must contain fully qualified domain names. The fully qualified domain name must be in the same domain.

ForwardAwareBaseUrlBuilder

This implementation parses `HTTP Host`, `Forwarded` and `X-Forwarded-*` headers to determine proxy information for the base URL.

Note: Be aware that these headers are vulnerable to malicious client-side XSS attacks. You must use this plugin only in deployments where these security risks are accepted or mitigated.

Claim number generator plugin

The `IClaimNumGenAdapter` claim number generator plugin is responsible for generating a claim number. There are two plugin methods that you must define: `generateNewClaimNumber` and `generateTempClaimNumber`. Each method must generate a unique claim number.

The `generateTempClaimNumber` can generate a different format of claim number appropriate for temporary use. For example, if you create the claim but it is not yet complete, the claim number is temporary. If the claim data is complete and soon opens, ClaimCenter calls the plugin method `generateNewClaimNumber` for its permanent claim number. Both methods take a `Claim` entity instance as the only argument, and return a claim number as a `String`.

ClaimCenter includes a feature called field validators that help you ensure accurate input within the web application user interface. In the case of claim numbers, field validators ensure that user input of a claim number exactly matches the correct format. For example, the following example shows a claim number field validator.

```
<ValidatorDef name="ClaimNumber" value="[0-9]{3}-[0-9]{5}"
  description="Validator.ClaimNumber"
  input-mask="###-####" />
```

If you implement claim number generator code, remember to update the claim number field validator so that it matches the actual format of your claim numbers. Failure to update the field validator can cause data entry of claim number to fail.

Typically, ClaimCenter calls this plugin once for each claim. However, there are rare cases in which the new claim number generator may get called twice for the same claim. Such a case occurs whenever the claim gets a validation error, the user then logs out of ClaimCenter, locates the draft of the failed claim, and then saves it again.

Cancelling claim numbers

There is also a method called `cancelNewClaimNumber` which you must implement. However, it is acceptable to have the method do nothing. This method gives plugin implementors a chance to cancel a previously allocated claim number. It is called in the rare case that the `generateNewClaimNumber` method was called to allocate a claim number but the claim fails validation and is subsequently canceled or otherwise unfinished. In this case, ClaimCenter calls the `cancelNewClaimNumber` method to give the claim number generator chance to reuse the previously-allocated number. This method takes a `Claim` entity instance and the claim number as a `String`.

Even if you implement `cancelNewClaimNumber` and perform some intelligent reacquiring algorithm for the number, a small chance remains for losing a claim number. For example, a power failure after allocating a claim number from external systems but before a ClaimCenter database commit.

Approval plugin

The approval plugin implementation answers two questions.

- Does the set of transactions need approval?
- If so, who must give that approval?

If no external plugin is implemented, ClaimCenter answers the first question by comparing the transaction set to the user's authority limits as described in the *Administration Guide*. If approval is required, then ClaimCenter consults the Approval rule set to answer the second question.

Your approval plugin must have two methods that correspond to these two questions.

- `requiresApproval` – Determines whether the user has any authority and, if so, whether the user has sufficient authority. The `ApprovalResult` object returns these two answers along with a list of messages indicating why approval is required.
- `getApprovingUser` – Determines the user and group to assign the approval activity.

Automatic address completion and fill-in plugin

To customize automatic address completion, you can create a class that implements the `IAddressAutocompletePlugin` plugin interface.

In the base configuration, the class `DefaultAddressAutocompletePlugin` implements this interface. The class provides the default behavior which uses `address-config.xml` and `zone-config.xml` files.

You can write your own plugin implementation if you want to handle address auto-completion and auto-fill differently. For example, you might want to access an address data service directly instead of having to import zone data files.

See the Javadoc for `IAddressAutocompletePlugin` for more information on this plugin interface.

Phone number normalizer plugin

ClaimCenter supports multiple fields for phone numbers. Each phone number type has a country code, a phone number, and an extension. The country code is a typekey to the `PhoneCountryCode` typelist, which is a list of regions and their regional phone codes.

ClaimCenter provides a plugin interface, `IPhoneNormalizerPlugin`, that you can use to customize the format of phone numbers. In the base configuration, the default implementation of the plugin is `gw.plugin.phone.CCPhoneNormalizerPlugin`.

The interface includes the following method signatures:

- `isPossibleNumber(String) : boolean`
- `isPossibleNumberWithExtension(String) : boolean`
- `normalizeNumberIfPossible(String) : String`
- `parsePhoneNumber(String) : GWPhoneNumber`
- `formatPhoneNumber(GWPhoneNumber number) : String`
- `normalizePhoneNumbersInBean(KeyableBean) : void`
- `normalizePhoneNumbersInArchive(IArchivedEntity, java.util.List<PhoneColumnProperties>) : void`

The plugin is called by the Phone Number Normalizer work queue. The plugin is also called whenever an entity containing a phone number is modified in ClaimCenter or is restored from the archive.

If you add new phone fields on existing objects or extension objects, customize or extend the plugin implementation to handle the additional phone fields. The `gw.api.util.PhoneUtil` class contains helper methods to facilitate formatting and parsing phone number records.

Define both `isPossibleNumber` and `isPossibleNumberWithExtension` methods to be very loose, non-country specific, validations. These methods essentially just need to check if the passed string could be a number in any country.

In the default phone normalizer plugin implementation, if `isPossibleNumber` returns `true`, the `normalizeNumberIfPossible` method strips all decorator and formatting characters from the number. The normalizer ignores all numeric characters as well as + and * characters. The plugin normalizes a phone number only if `isPossibleNumber` returns `true`. If `isPossibleNumber` returns `true`, the plugin calls `parsePhoneNumber` to convert the number to a `GWPhoneNumber` object.

By default, the length of a phone number extension field is 4. You can change the length of phone number extensions by specifying an `extensionLength` parameter on the plugin implementation. You can also separately set the maximum extension length in the `parsingExtensionLength` parameter. This value is typically used by the parser to enable a user who enters a longer than valid phone extension to get an appropriate error message. By default, this number is 7.

To change these values:

1. In Guidewire Studio™, open **configuration > config > Plugins > registry > IPhoneNormalizerPlugin.gwp**.
2. Click the **Add Parameter**  icon next to **Parameters**.
3. Enter `extensionLength` for the **key**.
4. Enter a numeric value for **value**.
5. Click the **Add Parameter**  icon next to **Parameters**.
6. Enter `parsingExtensionLength` for the **key**.
7. Enter a numeric value for **value**.

IMPORTANT: If you change the value of `parsingExtensionLength` and you are using Solr, also set the `parsingExtensionLength` field on the phone number analyzer defined in the Solr `schema.xml` file, as described in the following code sample.

For example, in Guidewire Studio, navigate in the **Project** window to **configuration > config > solr > claimcontact** and open `schema.xml`. If you set `parsingExtensionLength` to 10 for `IPhoneNormalizerPlugin`, set `parsingExtensionLength` as follows in the `solr.TextField` field type named `gw_phone`:

```
<fieldType name="gw_phone" class="solr.TextField"
    omitNorms="true" omitTermFreqAndPositions="false">
    <analyzer type="index">
        <tokenizer class="solr.KeywordTokenizerFactory"/>
        <filter class="solr.ASCIIFoldingFilterFactory"/>
        <filter class="com.guidewire.solr.analyzers.phone.PhoneFilterFactory"
            region="US"
            parsingExtensionLength="10"/>
    </analyzer>
    <analyzer type="query">
        <tokenizer class="solr.KeywordTokenizerFactory"/>
        <filter class="solr.ASCIIFoldingFilterFactory"/>
        <filter class="com.guidewire.solr.analyzers.phone.PhoneQueryFilterFactory"
            region="US"
            parsingExtensionLength="10"/>
    </analyzer>
</fieldType>
```

See also

- “Phone Number Normalizer work queue” in the *Administration Guide*

Official IDs mapped to tax IDs plugin

Contacts in the real world have many official IDs. However, only one of a contact's official IDs is used as a tax ID. In the default ClaimCenter data model, each `Contact` instance has a single `TaxID` field. Also, each `Contact` instance has an `OfficialIDs` array field. The array holds all sorts of official IDs for a contact, including the contact's official tax ID.

You configure ClaimCenter with official ID types in the `OfficialID` typelist. For example, the base configuration includes type keys for the U.S. Social Security Number (SSN) and the U.S. Federal Employer Identification Number (FEIN). The `OfficialIDs` array on a `Contact` instance contains instances of the `OfficialID` entity type, which has a field for the `OfficialIDType` typekey of the instance.

Create a Gosu implementation of the `OfficialIdToTaxIdMappingPlugin` interface to configure which typekeys from the `OfficialID` typelist you want to treat as official tax IDs. The plugin interface has one method, `isTaxId`, which takes a typekey from the `OfficialIDType` typelist (`oIdType`). The method returns `true` if you want to treat that official ID type as a tax ID. The default Java implementation that ClaimCenter provides always returns `false`.

The following sample Gosu implementation of the `OfficialIdToTaxIdMappingPlugin` interface returns `true` for official ID typekeys that have `SSN` or `FEIN` as codes.

```
uses gw.plugin.contact.OfficialIdToTaxIdMappingPlugin
class MyOfficialIdToTaxIdMappingPlugin implements OfficialIdToTaxIdMappingPlugin {

    /**
     * Return true if the official ID type is either an SSN or a FEIN.
     */
    override function isTaxId(oIdType : OfficialIDType) : boolean {
        return OfficialIDType.TC_SSN == oIdType or OfficialIDType.TC_FEIN == oIdType
    }
}
```

See also

- *ClaimCenter Application Guide*

Testing clock plugin (for non-production servers only)

It is possible modify the ClaimCenter server time either from the user interface or programmatically with the `ITestingClock` plugin. The ability to change the time is provided for testing ClaimCenter behavior over a long, simulated period of time. Guidewire supports changing the server time for testing purposes on non-production development servers only.

Certain areas in ClaimCenter are not affected by the modified time.

- The time shown in the next scheduled run for batch processes
- Today's date on the `Calendar` user interface

Working with the testing clock

Do not do the following on a production server:

- Do not attempt to manually change the testing clock on a production server.
- Do not attempt to call the `ITestingClock` plugin on a production server.

Do not attempt to go backwards in time using the testing clock. Time must always advance and never go backward. Guidewire does not support setting the testing clock to go back in time. Any attempt to do so can cause unpredictable behavior.

Guidewire does not recommend that you create your own custom implementation class for the `ITestingClock` plugin. Instead, use the following plugin implementation class as it is the most commonly used class and it contains safeguards to prohibit backdating of the clock:

`OffsetTestingClock`

This implementation class handles time synchronization of the nodes across the application cluster automatically.

The plugin implementation class

The `OffsetTestingClock` plugin interface has two methods—`getCurrentTime` and `setCurrentTime`—which get and set the current time using the standard ClaimCenter format of milliseconds stored in a long integer.

If it is not possible to set the time in the `setCurrentTime` function—for example, if you are using an external time server, and it is temporarily unreachable—the code throws exception `java.lang.IllegalArgumentException`.

After the application calls `setCurrentTime`, the time advances for any code that calls the `DateUtil` utility class to obtain the current date. The `setCurrentTime` method affects all ClaimCenter users, not just the user that called the method.

```
var d = gw.api.util.DateUtil.CurrentDate
```

For Gosu configuration code to be compatible with the testing clock, it must always obtain the date with `gw.api.util.DateUtil.CurrentDate` rather than other APIs. For example, do not use the standard Java class `java.util.Date` with the testing clock as the class returns the actual server time and not the ClaimCenter time.

Using the testing clock

The testing clock is available from the ClaimCenter user interface or the `ITestingClock` plugin.

Enable the testing clock

Procedure

1. Start Guidewire Studio.
2. In the **Project** window, navigate to **configuration > config > Plugins > registry**.
3. If `ITestingClock` does not exist in the registry, create the plugin by performing the following steps:
 - a) Right-click **registry** and click **New > New Plugin**.
 - b) In the **Name** field, type `ITestingClock`.
 - c) In the **Interface** field, type `ITestingClock`.
 - d) Click **OK**.
4. If necessary, set the plugin class to the internal offset testing clock by performing the following steps:
 - a) In the Plugins Registry editor, click **Add Plugin** , and then click **Add Java Plugin**.
 - b) In the **Java Class** field, type the following class name.

```
com.guidewire.pl.plugin.system.internal.OffsetTestingClock
```
5. In the **Project** window, navigate to **configuration > config**, and then open `config.xml`.
6. In `config.xml`, set `EnableInternalDebugTools` to `true`. If you see a message asking if you want to edit the file, click **Yes**.

```
<!-- Enable internal debug tools page http://localhost:8080/app/InternalTools.do
     Shared with CM
-->
<param name="EnableInternalDebugTools" value="true"/>
```
7. Save your changes.
8. Stop, and then restart ClaimCenter.

Access the testing clock from the user interface

Procedure

1. If you are operating a cluster of ClaimCenter servers, shut down the servers until only a single server is running.
2. Log in as user **su** with password **gw**.
3. To advance the clock a fixed number of days, enter the following text in the **Quick Jump** field on the upper right and then click **Go**. Change the number of days from 10 to the desired number of days. Afterward, a message shows the new current date in the yellow area near the top of the screen.

```
Run Clock addDays 10
```

4. Start up the other servers.

Startable plugins

A startable plugin is registered custom code that begins executing when a specified server run level is reached during server startup. Unlike a standard plugin, which executes only when it is invoked by other code, a startable plugin can be started and stopped as required.

A startable plugin can be used for various purposes.

- As a daemon, such as a listener-type plugin to a JMS queue.
- Periodic batch processing. For example, a batch process plugin to delete expired files from the file system.
- To initialize data, such as a plugin to load configuration data into the application database.

A listener-type startable plugin is similar to a message reply plugin which processes a reply acknowledgment for a sent message. The difference between the two types of plugins is that a startable plugin begins executing during server startup, while a message reply plugin is implemented as a callback method.

Two types of startable plugins are supported.

- A Singleton startable plugin runs in a single instance on a single server in the cluster. The server must have the `startable` server role. A Singleton startable plugin can begin executing at either the `DAEMONS` or `MULTIUSER` server run levels.
- A Distributed startable plugin runs on all servers in the cluster. A Distributed startable plugin can begin executing at any server run level. For example, a Distributed startable plugin can be used to perform early configuration validation by beginning execution at the `STARTING` run level.

A Distributed startable plugin is defined by specifying the `@Distributed` annotation on the plugin's class declaration. If the `@Distributed` annotation is not specified, the defined plugin is a Singleton startable plugin.

Starting a startable plugin

A startable plugin begins executing when a specified server run level is reached during server startup. The server run level is specified in the `@Availability` annotation on the plugin's class declaration. The annotation accepts an argument of type `AvailabilityLevel` which specifies the server run level, such as `DAEMONS` or `MULTIUSER`. The `@Availability` annotation is applicable for startable plugins only. Other types of plugins are executed when they are invoked by application code and are independent of the server run level.

By default, a Distributed startable plugin begins executing when the server reaches the `NODAEMONS` server run level, also called the maintenance system run level. To start the plugin at a different run level, use the `@Availability` annotation. A Distributed startable plugin can begin executing at any server run level.

A Singleton startable plugin must explicitly specify its server run level in an `@Availability` annotation. A Singleton startable plugin can begin executing at either the DAEMONS or MULTIUSER server run levels.

The following example code defines a Singleton startable plugin that begins executing at the MULTIUSER run level.

```
uses gw.api.server.Availability
uses gw.api.server.AvailabilityLevel

@Availability(AvailabilityLevel.MULTIUSER)
class HelloWorldStartablePlugin implements IStartablePlugin {
    ...
}
```

Whenever a Singleton plugin is started on a server that is not clustered, the plugin is started synchronously when the relevant server run level is reached. A Singleton plugin on a clustered server is started asynchronously, which results in the possibility that the plugin will start slightly after the relevant server run level is reached.

For a startable plugin running on a clustered server, certain events, such as a load balancing request or a failure, can cause it to be spontaneously moved around the cluster. During this time, the plugin can become temporarily unavailable.

Initializing a startable plugin

To initialize a startable plugin, the following plugin methods are called.

- `construct` – Constructor called when the plugin object is created
- `start` – Called to set up the plugin for subsequent execution. The `start` method contains an `execute` callback method which performs the actual plugin operations.

The initialization methods set up the plugin for its subsequent operations which are performed in the `execute` callback method. The initialization methods must not throw an exception. If exception-throwing initialization code is required then best practice locates such code in the `execute` callback method.

Registering startable plugins

Register your startable plugin implementation in the Plugins Registry in Studio, similar to how you register standard plugin implementations. In the **Project** window, navigate to **configuration > config > Plugins > registry**. Right-click on registry, and select **New > Plugin**.

In the application user interface, all registered startable plugin implementations are listed on the **Server Tools > Startable Services** page.

If the plugin's **Disabled** checkbox is selected in Studio, the plugin is never started and never appears in the **Startable Services** user interface.

Manually starting and stopping a startable plugin

If you have administration privileges, you can view the operational status of registered startable plugins on the **Server Tools > Startable Services** page. In the **Action** column, use the **Start** and **Stop** buttons to start and stop the service that a startable plugin provides. You can modify the PCF files for the **Startable Services** page to show additional information about your startable plugins, their underlying transport mechanisms, or any other information.

If a server running a startable plugin is terminated in a nonstandard manner then the resources associated with the plugin may not be appropriately released. Examples of nonstandard server terminations include power outages and forcible terminations.

See also

- For information on using a web service to start and stop startable plugins, see “Using web service methods with startable plugins” on page 127.

Writing a singleton startable plugin

A Singleton startable plugin runs on a single server in the cluster.

Write a new class that implements the `IStartablePlugin` interface. Implement the following methods.

- A `start` method to start your service.
- A `stop` method to stop your service.
- A property accessor function to get the `State` property from your startable plugin.

This method returns a typecode from the typelist `StartablePluginState`: the value `Stopped` or `Started`. The administration user interface uses this property accessor to show the state to the user. Define a private variable to hold the current state and your property accessor (`get`) function can look simple.

```
// Private variables...
var _state = StartablePluginState.Stopped;

...
// Property accessor (get) function...
override property get State() : StartablePluginState {
    return _state // return our private variable
}
```

Alternatively, combine the variable definition with the shortcut keyword to simplify your code. You can combine the property definition with the variable definition in a single code statement.

```
var _state : StartablePluginState as State
```

The plugin includes a constructor which is called on system startup. However, locate the service code in the plugin's `start` method, not its constructor.

The `start` method must initialize the plugin's `State` property by using an internal variable defined in the plugin. The variable's value must be one of the supported values of the `StartablePluginState` class, such as `Started` or `Stopped`.

The `start` method can also start any desired listener code or threads, such as a JMS queue listener.

The `start` method accepts an argument that references a callback handler of type `StartablePluginCallbackHandler`. This callback is important if the startable plugin modifies any entity data, which most startable plugins do.

Any plugin code that affects entity data must be run within a method called `execute`. The `execute` method accepts an argument of a special type of in-line function called a Gosu block, which is described in the *Gosu Reference Guide*. The Gosu block itself accepts no arguments. The `execute` method is then passed to the callback handler.

If you do not need a user context, use the simplest version of the callback handler method `execute`, whose one argument is the Gosu block.

You do not need to create a separate bundle. If you create new entities to the current bundle, they are in the correct default database transaction the application sets up for you. Use the code returned by the `Transaction.getCurrent` method to get the current bundle if you need a reference to the current writable bundle.

The Java language does not directly support blocks. If you implement your plugin in Java, you cannot use a Gosu block but you can use an anonymous class to do the same thing.

The plugin's `start` method also includes a boolean variable that indicates whether the server is starting. If `true`, the server is starting up. If `false`, the start request comes from the `Startable Services` user interface.

The following shows a simple Gosu block that changes entity data. This example assumes your listener defined a variable `messageBody` with information from your remote system. If you get entities from a database query, remember to add them to the current bundle.

The following example queries all User entities and sets a property on results of the query.

```
// Variable definition earlier in your class...
var _callback : StartablePluginCallbackHandler;

...

override function start( cbh: StartablePluginCallbackHandler, isStarting: boolean ) {
    _callback = cbh
    _callback.execute( \ -> {

        var q = gw.api.database.Query.make(User) // run a query
        var b = gw.Transaction.Transaction.Current // get the current bundle

        for (e in q.select()) {
            // Add entity instance to writable bundle, then save and only modify that result
            var writable_object = bundle.add(e)

            // Modify properties as desired on the result of bundle.add(e)
            writable_object.Department = "Example of setting a property on a writable entity instance."
        }
    })
    // You do not need to commit the bundle here. The execute method commits after your block runs.
}
```

Note that to make an entity instance writable, save and use the return result of the bundle add method.

Just like your `start` method must set your internal variable that sets its state to started, your `stop` method must set your internal state variable to `StartablePluginState.Stopped`. Additionally, stop whatever background processes or listeners you started in your `start` method. For example, if your `start` method creates a new JMS queue listener, your `stop` method destroys the listener. Similar to the `start` method's `isStartingUp` parameter, the `stop` method includes a `boolean` variable that indicates whether the server is shutting down now. If `true`, the server is shutting down. If `false`, the stop request comes from the **Startable Services** user interface.

User contexts for startable plugins

If you use the simplest method signature for the `execute` method on `StartablePluginCallbackHandler`, your code does not run with a current ClaimCenter user. Any code that directly or indirectly runs due to this plugin—including pre-update rules or any other code—must be prepared for the current user to be `null`. You must not rely on non-null values for current user if you use this version.

However, there are alternate method signatures for the `execute` method. Use these to perform your startable plugin tasks as a specific User. Depending on the method variant, pass either a user name or the actual `User` entity.

On a related note, the `gw.transaction.Transaction` class has an alternate version of the `runWithNewBundle` method to create a bundle with a specific user associated with it. You can use this in contexts in which there is no built-in user context or you need to use different users for different parts of your tasks.

```
gw.transaction.Transaction.runWithNewBundle(\ bundle -> YOUR_BLOCK_BODY, user)
```

For the second method argument to `runWithNewBundle`, pass either a `User` entity or a `String` that is the user name.

Simple startable plugin example

The following sample code implements a complete Singleton startable plugin. The example does not do anything useful in its `start` method, but demonstrates the basic structure of creating a block that you pass to the callback handler's `execute` method.

```
package gw.api.startableplugin
uses gw.api.startable.IStartablePlugin
uses gw.api.startable.StartablePluginCallbackHandler
uses gw.api.startable.StartablePluginState
uses gw.api.server.Availability
uses gw.api.server.AvailabilityLevel

@Availability(AvailabilityLevel.MULTIUSER)
class HelloWorldStartablePlugin implements IStartablePlugin {
    var _state = StartablePluginState.Stopped;
    var _callback : StartablePluginCallbackHandler;

    construct() {
```

```

    }

    override property get State() : StartablePluginState {
        return _state
    }

    override function start( cbh: StartablePluginCallbackHandler, isStarting: boolean ) {
        _callback = cbh
        _callback.execute( \ -> {
            // Do some work
            // [...]
        } )
        _state = StartablePluginState.Started
        _callback.log( "*** From HelloWorldStartablePlugin: Hello world." )
    }

    override function stop( isShuttingDown: boolean ) {
        _callback.log( "*** From HelloWorldStartablePlugin: Goodbye." )
        _callback = null
        _state = StartablePluginState.Stopped
    }
}

```

Writing a distributed startable plugin

A Distributed startable plugin runs on every server in a cluster. Guidewire strongly recommends that Distributed plugins save their current start/stop state in a database. Persisting the start/stop state enables the plugin to handle edge cases, such as a server joining the cluster after other servers have started. To keep the state consistent across all servers, the state must persist in the database. Custom code can be written to persist the plugin's start/stop state.

The events related to servers and startable plugins are described in the following scenarios.

1. The server starts. When the appropriate server run level is reached, the startable plugin's `start` method is invoked with the `serverStartup` argument set to `true`.
2. Operations enacted through the user interface or an invoked API can result in a request being sent to a server to start or stop a particular startable plugin. The server processes the request by calling the plugin's `start` or `stop` method. In these cases, the `start` method's `serverStartup` argument is set to `false`. Similarly, the `stop` method's `serverStopping` argument is also set to `false`.
3. The server shuts down. The startable plugin's `stop` method is invoked with the `serverStopping` argument set to `true`.

The plugin's `start` method accepts a handler argument of the type `StartablePluginCallbackHandler`. The methods implemented in the handler manage the database state information.

- `getState` – Retrieves the current start/stop state from the database. Returns `true` if the server has started, otherwise `false`. The first time the plugin is run on a server, the database state field does not yet exist. To handle this situation, the method accepts a single argument that specifies the default state to use to initialize the field. If the database field has been set by earlier operations, the argument is ignored.
- `setState` – Sets the plugin's start/stop state in the database. The method accepts a single argument that specifies the current state: `true` if running, otherwise `false`.
- `logStart` – Logs the event of starting the plugin. The method accepts a single argument that specifies the text describing the event.
- `logStop` – Logs the event of stopping the plugin. The method accepts a single argument that specifies the text describing the event.

The following Gosu example code implements `start` and `stop` methods for a Distributed startable plugin. It creates a trivial thread and maintains the plugin's start/stop state.

```

package gw.api.startableplugin
uses gw.api.startable.IStartablePlugin
uses gw.api.startable.StartablePluginCallbackHandler
uses gw.api.startable.StartablePluginState
uses gw.transaction.Transaction
uses java.lang.Thread

@Distributed
class HelloWorldDistributedStartablePlugin implements IStartablePlugin {

```

```

var _state : StartablePluginState
var _startedHowManyTimes = 0
var _callback : StartablePluginCallbackHandler
var _thread : Thread

override property get State() : StartablePluginState {
    return _state
}

override function start(handler : StartablePluginCallbackHandler,
                       serverStartup : boolean) : void {
    // Save the callback handler
    _callback = handler

    // Is server starting?
    if (serverStartup) {
        // Plugin is starting, too
        _state = _callback.getState(Started)

        // Log event
        if (_state == Started) {
            _callback.logStart("HelloWorldDistributedStartablePlugin:Started")
        } else {
            _callback.logStart("HelloWorldDistributedStartablePlugin:Stopped")
        }
    } else {
        // Plugin start/stop state has changed
        _state = Started
        if (_callback.State != Started) {
            changeState(Started) // Update saved start/stop state
        }
        _callback.logStart("HelloWorldDistributedStartablePlugin")
    }

    // If thread already exists, stop it before restarting it
    if (_state == Started) and (_thread != null) {
        _thread.stop()
    }

    // Increment local counter
    _startedHowManyTimes++

    // Create and start thread
    var t = new Thread() {
        function run() {
            print("hello!")
        }
        t.Daemon=true
    }
}

override function stop(serverStopping : boolean) : void {
    // Stop thread
    if (_thread != null) {
        _thread.stop()
        _thread = null
    }

    if (_callback != null) {
        if (serverStopping) {
            if (_state == Started) {
                _callback.logStop("HelloWorldDistributedStartablePlugin:Started")
            } else {
                _callback.logStop("HelloWorldDistributedStartablePlugin:Stopped")
            }
            _callback = null
        } else {
            if (_callback.State != Stopped) {
                changeState(Stopped) // Update saved start/stop state
            }
            _callback.logStop("HelloWorldDistributedStartablePlugin")
        }
        _state = Stopped
    }

    // Internal function to set the database state. If an exception occurs, retries 5 times.
    private function changeState(newState : StartablePluginState) {
        var tryCount = 0
        while (_callback.State != newState && tryCount < 5) {
            try {
                Transaction.runWithNewBundle(\ bundle -> { _callback.setState(bundle, newState)},
                                              User.util.UnrestrictedUser)
            }
            catch (e : java.lang.Exception) {
                tryCount++
                _callback.log(this.IntrinsicType.Name + " on attempt " + tryCount +
                            " caught " + (typeof e).Name + ":" + e.Message)
            }
        }
    }
}

```

```
}
```

Defining startable plugins in Java

You can develop your custom startable plugin in Java, but special considerations apply.

If you have Java files for your startable plugin, place your Java class and libraries files in the same places as with other plugin types.

In Gosu, your startable plugin must call the `execute` method on the callback handler object, as discussed in previous topics.

```
override function start( cbh: StartablePluginCallbackHandler, isStarting: boolean ) : void {
    _callback = cbh
    _callback.execute( \ -> {
        //...
    })
}
```

However, the Java language does not directly support blocks. If you implement your plugin in Java, you cannot use a Gosu block. However, instead you can use an anonymous class.

From Java, the method signatures for the `execute` methods (there are multiple variants) take a `GWRunnable` for the block argument. `GWRunnable` is a simple interface that contains a single method, called `run`. Instead of using a block, you can define an in-line anonymous Java class that implements the `run` method. This is analogous to the standard Java design pattern for creating an anonymous class to use the standard class `java.lang.Runnable`.

```
GWRunnable myBlock = new GWRunnable() {
    public void run() {
        System.out.println("I am startable plugin code running in an anonymous inner class");
        // Add more code here...
    }
};

_callbackHandler.execute(myBlock);
```

Persistence and startable plugins

Your startable plugin can manipulate Guidewire entity data. If your startable plugin needs to maintain state for itself, perform one of the following actions.

- Create your own custom persistent entity types that track the internal state information of your startable plugin.
- Use the system parameter table for persistence.

Multi-threaded inbound integration

ClaimCenter provides a plugin interface that supports high performance multi-threaded processing of inbound data. ClaimCenter includes default implementations for the most common usages: reading text file data and receiving Java Message Service (JMS) messages. If these integrations do not serve your needs, you can write your own integration based on the plugin interfaces in the `gw.plugin.integration.inbound` package.

Some requirements for high-performance data throughput for inbound integrations require special threading or transaction features from the hosting J2EE/JEE application environment. Writing correct, thread-safe, high-performing code that interacts with the application server's transactional facilities is difficult. ClaimCenter includes tools that help you write such inbound integrations. You can focus on your own business logic rather than on how to write thread-safe code to run in each application server.

You can test your integration code on the QuickStart server that runs from Guidewire Studio. This server runs in `dev` mode in the Jetty environment. Running inbound integrations on Jetty is not supported in any other configuration.

Inbound integration configuration XML file

The configuration file for inbound integrations is `inbound-integration-config.xml` in the `configuration/config/integration` folder. Edit this file in Studio to define configuration settings for every inbound integration that you use. Each inbound integration requires configuration parameters such as references to registered plugin implementations.

Making import actions repeatable without side effects or errors

If errors happen part way through processing a file, some lines in the file might be processed again when ClaimCenter retries the process. To ensure recovery from errors in the middle of an import, particularly for files, design your import data formats and code so that requests can be repeated without side effects. This quality is formally known as *idempotency*.

For example, if the inbound integration creates new imports with a specified unique ID, your code can first check if the record already exists. If it already exists, your code could re-apply that request or skip that record without generating errors or creating duplicate records.

Inbound integration core plugin interfaces

ClaimCenter provides the multi-threaded inbound integration system in plugin interfaces for two use cases. Each interface defines a contract between ClaimCenter and inbound integration high-performance multi-threaded processing of input data. The interfaces are in the `gw.plugin.integration.inbound` package.

InboundIntegrationMessageReply

Inbound high-performance multi-threaded processing of replies to messages that a `gw.plugin.messaging.MessageTransport` implementation sends. This interface is a subinterface of `gw.plugin.messaging.MessageReply`.

If your code throws an exception, the transaction of the message processing is rolled back. The original message is restored to the queue.

InboundIntegrationStartablePlugin

Inbound high-performance multi-threaded processing of input data as a startable plugin. Use a startable plugin for all contexts other than handling replies to messages that a `MessageTransport` implementation sends. This interface is a subinterface of `gw.api.startable.IStartablePlugin`.

If your code throws an exception, ClaimCenter uses the value of the `stoponerror` parameter in the configuration file to determine what action to take. If that value is `true`, ClaimCenter performs error handling that is applicable to the integration type. If the value of `stoponerror` is not `true`, ClaimCenter skips the item that caused the exception. Ensure that your code logs any errors or notifies an administrator.

For some use cases, you do not need to write your own implementation of these main plugin interfaces. ClaimCenter includes plugin implementations that support common use cases and that are supported for production servers. Both file and JMS plugin implementations are provided in variants for message reply and startable plugin use.

ClaimCenter supports the following types of inbound integrations:

File inbound integrations

Use this integration to read text data from local files. Poll a directory in the local file system for new files at a specified interval. Send new files to integration code and process incoming files line by line, or file by file. You provide your own code that processes one chunk of work, either one line, or one file, depending on how you configure it. Your code is called a handler plugin.

JMS inbound integration

Use this integration to get objects from a JMS message queue. You provide your own code that processes the next message on the JMS message queue. Your code is called a handler plugin.

Custom integration

If you process incoming data other than files or JMS messages, write your own version of the `InboundIntegrationMessageReply` or `InboundIntegrationStartablePlugin` plugin interface. In both cases, for custom integrations you must write multiple classes that implement helper interfaces such as `WorkAgent`.

You can implement your plugin code using any of Gosu, Java with no OSGi, or Java as an OSGi bundle. If you use Java or if you require third-party libraries, Guidewire recommends implementing your code as an OSGi plugin.

In all cases, you must register and configure plugin implementations in the Studio Plugins Registry. See each topic for more information about which implementation classes to register. Additionally, for file and JMS integrations, you write handler classes.

The definition of a plugin implementation for an inbound integration in the Plugins Registry includes a plugin parameter called `integrationservice`. That `integrationservice` parameter defines how ClaimCenter finds configuration information in the inbound integration configuration XML file.

Inbound integration handlers for file and JMS integrations

Whether you write your own integration or use the integrations that the base configuration provides, you must write code that handles one chunk of data. In both cases, the interface implementation that you use depends on whether you are using messaging.

To write a custom integration that supports data other than files or JMS messages, your code primarily implements the interface `InboundIntegrationMessageReply` or `InboundIntegrationStartablePlugin`.

If you use the file or JMS inbound integrations that the base configuration provides, you register an existing plugin implementation of `InboundIntegrationMessageReply` or `InboundIntegrationStartablePlugin`. ClaimCenter includes plugin implementations of those interfaces to process files or JMS data.

You can implement your handler plugin code using either Gosu, Java with no OSGi, or Java as an OSGi bundle. If you use Java or if you require third-party libraries, Guidewire recommends implementing your code as an OSGi plugin.

For file or JMS inbound integrations, you must write a handler class that processes one chunk of data. ClaimCenter defines a handler plugin interface called `InboundIntegrationHandlerPlugin` that contains one method called `process`. The file or JMS integration process calls the method to process one chunk of data. The method has no return value. ClaimCenter passes the chunk of data to the method as a parameter of type `java.lang.Object`. In your implementation of the handler class, you write the `process` method. Downcast the `Object` to the necessary type before using it.

- For file handling, the data type depends on the value you set for the `processingmode` parameter.
 - If you set the `processingmode` parameter to `line`, downcast the `Object` to `String`.
 - If you set the `processingmode` parameter to `file`, downcast the `Object` to `java.nio.file.Path`.
- For JMS handling, downcast the `Object` to a JMS message of type `javax.jms.Message`.

Register the plugin in the Studio Plugins Registry. You must add one plugin parameter called `integrationService`. That plugin parameter links your plugin implementation to one XML element within the `inbound-integration-config.xml` file.

If you use either the file or the JMS variant of the startable plugin, your class must implement the `InboundIntegrationHandlerPlugin` interface. This interface specifies only the `process` method.

If you use either the file or the JMS variant of the message reply plugin, your class must implement the `InboundIntegrationMessageReplyHandler` interface, which is a subinterface of `InboundIntegrationHandlerPlugin`. Implement the `process` method and all methods of the `MessageReply` plugin, which are `initTools`, `suspend`, `shutdown`, `resume`, and `setDestinationID`. Save the parameters to your `initTools` method into private variables. Use those private variables during your `process` method to find and acknowledge the original `Message` object.

See also

- “Custom inbound integrations” on page 323
- “Registering a plugin implementation class” on page 172

Configuring inbound integration

You use Studio to access the configuration file for inbound integrations. In the `Project` window, navigate to `configuration > config > integration`, and the open `inbound-integration-config.xml`. The file contains thread pool and inbound integration configuration settings.

The first section of the `inbound-integration-config.xml` file configures thread pools. The `<threadpools>` element contains a list of thread pool elements, each of which is a subtype of the abstract `<threadpool>` type.

In the base configuration of ClaimCenter, the `<threadpools>` element contains a list of subelements with pre-defined thread pool names.

The second section of the `inbound-integration-config.xml` file configures integrations. The `<integrations>` element contains a list of integration elements, each of which is a subtype of the abstract `<inbound-integration>` type. For example, if you need five different JMS inbound integrations, each listening to its own JMS queue, add five elements to this section of the file. For each inbound integration, define configuration parameters as subelements. Some of the parameters are required, and some are optional. For example, you must declare the name of the registered plugin implementations that correspond to your handler code.

In the base configuration of ClaimCenter, the `<integrations>` element contains a list of subelements with pre-defined inbound integration names.

See also

- “Thread pool configuration XML elements” on page 310

- “Inbound integration configuration XML elements” on page 311
- “Registering a plugin implementation class” on page 172

Thread pool configuration XML elements

The following element types for thread pools are subtypes of the abstract `<threadpool>` type:

`<j2ee-managed-threadpool>`

This type of thread pool supports JMS integration and JSR-236 concurrency. Use this type for running in IBM WebSphere, JBoss, or Oracle Weblogic environments. Use this type of thread pool for running in a JBoss environment. This type does not support an inbound JMS integration that runs in dev mode on the Jetty platform.

`<unmanaged-threadpool-cached>`

This type of thread pool supports reusing an available thread that is not in use and creating a new thread if one is not available.

`<unmanaged-threadpool-fixed>`

This type of thread pool uses a fixed number of threads.

`<unmanaged-threadpool-forkjoin>`

This type of thread pool uses the same number of threads as the number of CPUs that the JVM detects. Use this type for a self-managing default thread pool.

`<unmanaged-threadpool-single>`

This type of thread pool uses a single thread.

Some of these types contain attributes or subelements specific to the type.

The base configuration provides definitions for managed and unmanaged thread pools that use default attribute values. These definitions are for development use only and must not be used in a production environment. For best performance, create your own custom thread pool with a unique name and tune that thread pool for the specific work you need, such as your JMS work. Then, update the thread pool settings to include the JNDI name for your new thread pool.

For testing inbound JMS integration in dev mode on the Jetty platform, use an unmanaged thread pool, not a thread pool of type `j2ee-managed-threadpool`.

`<threadpool>` XML element attributes

The parent `<threadpool>` type provides the following attributes:

Attribute name	Type	Required	Default value	Description
name	string	•		A unique identifying name for this thread pool in the <code>inbound-integration-config.xml</code> file. The name attribute is used to specify the thread pool that an inbound integration uses.
disabled	boolean	false		Determines whether to disable this thread pool. Set to <code>true</code> to disable the thread pool.
env	string			Sets a configuration that is valid only for a specific server environment or set of server environments. You can specify multiple values for this attribute by using a comma-separated list.

`<j2ee-managed-threadpool>` XML element subelement

The J2EE managed thread pool XML element has the following subelement.

Subelement name	Type	Required	Default value	Description
jndi	string •		java:comp/DefaultManagedScheduledExecutorService	The JNDI name of the application server thread pool.

<unmanaged-threadpool-fixed> XML element subelement

The unmanaged, fixed thread pool XML element has the following subelement.

Subelement name	Type	Required	Default value	Description
size	positiveInteger •		10	The number of threads to use in your thread pool.

Varying the inbound integration settings

ClaimCenter provides the following means to vary configuration of inbound integration by system environment.

Inbound integration XML file

In your `inbound-integration-config.xml` file, each thread pool or integration element has a `disabled` attribute. If the value of that attribute is `true`, ClaimCenter does not enable the thread pool or integration.

If the value of that attribute is `false`, the action that ClaimCenter takes depends on the value of the optional `env` attribute. If you provide no value for this attribute, ClaimCenter enables the thread pool or integration. You can provide either a single value or list of comma-separated values for the `env` attribute. If you provide any value for the `env` attribute, ClaimCenter enables the thread pool or integration element only if a value in that `env` attribute matches the `env` system environment configuration setting.

For example, to use different JMS settings for development and for production, list two `<jms-integration>` elements. For one element, set the `env` attribute to `development`. For the other element, set the `env` attribute to `production`. For each element, set appropriate JMS configuration settings for that system environment.

Plugin property configuration in Plugins Registry

In the Plugins Registry, you must set the `integrationservice` plugin property in the user interface in one or more Plugins Registry files. In a single Plugins Registry file, you can optionally define the `integrationservice` plugin property multiple times with values that vary by system environment or server ID values.

You can use one or both of these techniques to vary the runtime behavior of the server based on the server environment.

To specify more than one environment for a given top-level element, you use comma-separated values for the `env` attribute. Although Guidewire supports the `env` attribute within elements of the `inbound-integration-config.xml` file, you cannot vary the configuration by server ID.

Inbound integration configuration XML elements

The following element types for inbound integrations are subtypes of the abstract `<inbound-integration>` type:

`<file-integration>`

File inbound integration

`<jms-integration>`

JMS inbound integration

`<custom-integration>`

Custom inbound integration, directly implementing the `InboundIntegrationStartablePlugin` or `InboundIntegrationMessageReply` interface

<inbound-integration> XML element attributes and subelements

The abstract inbound integration XML element has the following attributes.

Attribute name	Type	Required	Default value	Description
name	string •			<p>A unique identifying name for this inbound integration in the <code>inbound-integration-config.xml</code> file.</p> <p>The <code>name</code> attribute must match the value of the <code>integrationService</code> plugin parameter in the Plugins registry for all registered plugin implementations of any inbound integration interfaces. In the Plugins registry, add the plugin parameter <code>integrationService</code> and set to the value of this unique identifying name.</p>
disabled	boolean		false	Determines whether to disable this inbound integration. Set to <code>true</code> to disable the inbound integration.
env	string			Sets a configuration that is valid only for a specific server environment or set of server environments. You can specify multiple values for this attribute by using a comma-separated list.

The abstract inbound integration XML element has the following subelements, in the order shown.

Subelement name	Type	Required	Default value	Description
threadpool	string •			<p>The unique name of a thread pool as configured earlier in the file.</p> <p>For testing inbound JMS integration in dev mode on the Jetty platform, use an unmanaged thread pool, not a thread pool of type <code>j2ee-managed-threadpool</code>.</p>
pollinginterval	string •		60	The time interval in seconds between polls, though the algorithm interacts with the throttle interval and the ordered parameter.
throttleinterval	string •		60	The time interval in seconds after polling, though the algorithm interacts with the polling interval and the ordered parameter.
ordered	string •		true	<p>By default, the inbound file integration handles files in a single thread sequentially.</p> <p>To handle multiple files at a time in parallel in multiple threads on this server, set this value to <code>false</code> and ensure that the thread pool has at least two threads.</p> <p>For file inbound integration, the order of the processing of individual files in the directory is undefined. Never rely on the order being any particular order, such as alphabetic or creation date.</p>
stoponerror	string •		true	<p>If <code>true</code>, ClaimCenter stops the integration if an error occurs. For JMS and custom implementations, the behavior depends on the type of the implementation and on the value of the <code>ordered</code> parameter.</p> <p>If not <code>true</code>, ClaimCenter skips the item if an error occurs. Be sure to log any errors or notify an administrator.</p>
transactional	string •		false	Set this parameter to <code>true</code> if your plugin supports transaction rollback. In this case, your plugin must implement the <code>TransactionalWorkSetProcessor</code> interface in the <code>gw.api.integration.inbound.work</code> package.
osgiservice	string •		true	Always set to <code>true</code> . This parameter is for internal use. This value is independent of whether you choose to register your handler class as an OSGi plugin.

<file-integration> XML element subelements

The file inbound integration XML element has the following additional subelements, in the order shown.

Subelement name	Type	Required	Default value	Description
traceabilityidcreationpoint	string	•	INBOUND_INTEGRATION_FILE	The value is specified by the constants in gw.logging.TraceabilityIDCreationPoint. As well as the default value, the following value is also valid: <ul style="list-style-type: none">• OUTBOUND_MESSAGE_REPLY
pluginhandler	string	•		The name in the Plugins registry for an implementation of the InboundIntegrationHandlerPlugin plugin interface. The value is the .gwp file name, not the implementation class name.
processingmode	string	•	line	To process one line at a time, set to line. In your handler class in the process method, you must downcast to String. To process one file at a time, set to file. In your handler class in the process method, you must downcast to java.nio.file.Path.
incoming	string	•		The full path of the configured incoming events directory. The value of this element must not be empty.
processing	string	•		The full path of the configured processing events directory. The value of this element must not be empty.
error	string	•		The full path of the configured error events directory. The value of this element must not be empty.
done	string	•		The full path of the configured done events directory. The value of this element must not be empty.
charset	string	•	UTF-8	The character set that the inbound file uses. The value of this element must not be empty.
createdirectories	boolean	•	false	If true, ClaimCenter creates the incoming, processing, error, and done directories if they do not already exist. If errors occur that prevent creation of any directories, the server does not start. If false, ClaimCenter requires that all of these directories must already exist. If

Subelement name	Type	Required	Default value	Description
				any directories do not already exist, the server does not start. For better security, set to false.

<jms-integration> XML element subelements

The JMS inbound integration XML element has the following additional subelements, in the order shown.

Subelement name	Type	Required	Default value	Description
traceabilityidcreationpoint	string	•	INBOUND_INTEGRATION_JMS	The value is specified by the constants in gw.logging.TraceabilityIDCreationPoint. As well as the default value, the following value is also valid: <ul style="list-style-type: none"> • OUTBOUND_MESSAGE_REPLY
pluginhandler	string	•		The name in the Plugins registry for an implementation of the InboundIntegrationHandlerPlugin plugin interface. The value is the .gwp file name, not the implementation class name. The value of this element must not be empty.
connectionfactoryjndi	string	•		The application server configured JNDI connection factory. The value of this element must not be empty.
destinationjndi	string	•		The application server configured JNDI destination. The value of this element must not be empty.
user	string	•	The default value is an empty string.	Username for authenticating on the JMS queue.
password	string	•	The default value is an empty string.	Password for authenticating on the JMS queue.
durablesSubscription	string	•	The default value is an empty string.	Reserved for future use.
nolocal	boolean	•	false	Reserved for future use.
messageselector	string	•	The default value is an empty string.	Reserved for future use.
messagereceivetimeout	unsignedInt	•	15	The maximum time in seconds to wait for an individual JMS message.
batchlimit	unsignedInt	•	5	The maximum number of messages to receive in a poll interval.
jndi-properties	A sequence of <jndi-property> elements	•		A sequence of arbitrary JNDI properties that you define.

Subelement name	Type	Required	Default value	Description
jndi-property	keyvaluepair			An arbitrary JNDI property that you define. Define key and value subelements that contain key/value pairs.

<custom-integration> XML element subelements

The custom inbound integration XML element has the following additional subelements, in the order shown.

Subelement name	Type	Required	Default value	Description
traceabilityidcreationpoint	string	•	INBOUND_INTEGRATION_CUSTOM	The value is specified by the constants in <code>gw.logging.TraceabilityIDCreationPoint</code> . As well as the default value, the following values are also valid: <ul style="list-style-type: none"> • INBOUND_INTEGRATION_FILE • INBOUND_INTEGRATION_JMS • OUTBOUND_MESSAGE_REPLY
workagentimpl	string	•		The fully qualified name, including the package, of the <code>InboundIntegrationStartablePlugin</code> or <code>InboundIntegrationMessageReply</code> implementation class. If your class is a Gosu class, you must also include the suffix <code>.gs</code> . For example: <code>mycompany.integ.MyInboundPlugin.gs</code> . The value of this element must not be empty.
parameters	A sequence of parameter elements	•		A sequence of arbitrary parameters that you define. For example, you can use these parameters to store server names and port numbers. For each parameter, create a parameter subelement. The plugin interface has a <code>setup</code> method. These parameters are in the <code>java.util.Map</code> that ClaimCenter passes to that initialization method.
parameter	keyvaluepair			An arbitrary parameter that you define. Define key and value subelements that contain key/value pairs.

Using the inbound integration polling and throttle intervals

Two configuration parameters are available for coordinating when the inbound integration work begins: `pollinginterval` and `throttleinterval`.

The primary controller is the polling interval (`pollinginterval`). Additionally, you can use the throttle interval (`throttleinterval`) parameter to reduce the impact on the server load or external resources.

The inbound integration performs work in the following sequence:

1. At the beginning of the polling interval, the integration polls for new work and creates new work items but does not start processing the work items.
2. ClaimCenter begins working on the work items.

- If the ordered parameter is `true`, the work is ordered. In this case, the work happens in the same thread.
 - If the ordered parameter is `false`, the work is unordered. In this case, the work happens in separate additional threads with no guarantee of strict ordering.
3. After the current work is complete, the system determines how much time has elapsed and compares that value with the two interval parameters. The behavior is slightly different for ordered and unordered work.
- If the work is ordered, the server finishes the existing work. Next, the server checks the elapsed total time since last polling. If the elapsed time is greater than the sum of the `pollinginterval` and `throttleinterval`, the server polls for new work immediately. Otherwise, the server waits until the sum of the `pollinginterval` and `throttleinterval` has elapsed.
 - If the work is unordered, the same time check occurs. However, the time check happens immediately after new work is created but does not wait for the work to be done. The work proceeds in parallel threads. When the last work item in a batch completes, the session is closed.

Inbound file integration

The base configuration of ClaimCenter includes classes that support file-based input with high-performance multi-threaded processing. ClaimCenter provides two classes, one for processing message replies, and one as a startable plugin.

You cannot modify the code in the plugin implementation class in Studio, but you can use one or more instances of these integrations to work with your own file data.

Inbound file integration proceeds in the following stages.

1. In response to an inbound event that is specific to your requirements, your own integration code creates a new file in a specified incoming directory on the local file system.
2. The inbound file integration class polls the incoming directory at a specified interval and detects any new files since the last time checked.
The order of the processing of individual files in the directory is undefined. Never rely on the order being any particular order, such as alphabetic or creation date.
3. The inbound file integration class moves all found files to the processing directory, which stores inbound files in progress.
4. The inbound file integration class opens each new file using a specified character set. The default is UTF-8, but it is configurable.
5. The inbound file integration class reads one unit of work and dispatches it to your handler code. The `processingmode` subelement in the `inbound-integration-config.xml` file defines the type of data processed in one unit of work. If that subelement has the value `line`, ClaimCenter sends one line at a time to the handler as a `String` object. If that subelement has the value `file`, ClaimCenter sends the entire file to the handler as a `java.nio.file.Path` object.

If an exception occurs during processing, the plugin class moves the file to the error directory. The value of the `stoponerror` subelement does not affect this behavior. The file name is changed to add a prefix that includes the time of the error, which is expressed in milliseconds as returned from the Java time utilities. For example, if the file name `ABC.txt` has an error, it is renamed in the error directory with a name similar to `1864733246512.error.ABC.txt`.

To retry a failed inbound file, your inbound file integration class must move the file from the error directory back into the incoming directory for reprocessing.

6. After successfully reading and processing the complete file, the inbound file integration class moves the file to the done directory.
7. If there were any other files detected in this polling interval in “step 2”, the inbound file integration class repeats the process at “step 4”. Optionally, you can set the integration to operate on the most recent batch of files in parallel. For related information, see the `ordered` subelement.

8. The inbound file integration class waits until the next polling interval, and repeats this process starting at “step 3”.

Create an inbound file integration

ClaimCenter uses inbound file integration to read information from files that an external system creates. This integration can be either a message reply plugin or a startable plugin.

The base configuration of ClaimCenter provides implementations for the message reply plugin and the startable plugin for inbound file integration. You can use these plugins as a template for your own plugin. ClaimCenter also provides implementations for the message reply handler plugin and the startable handler plugin for inbound file integration. You can use these plugins as a template for your own handler plugin. You provide the handler class that implements your business logic.

Procedure

1. In the Project window, navigate to **configuration > config > integration**, and open the file `inbound-integration-config.xml`.
2. Configure the thread pools.

You can use a thread pool element in the base configuration as a template.
3. In the list of integrations, create one `<integration>` element of type `<file-integration>`.

You can use a `<file-integration>` element in the base configuration as a template.
4. Set configuration parameter subelements, ensuring that the order of subelements is correct.

The `inbound-integration-config.xsd` file specifies the correct sequence of the subelements.
5. Create the inbound file integration plugin.
 - a) In Studio, in the Plugins registry, add a new .gwp file.
 - b) Studio prompts for a plugin name and plugin interface. For the plugin name, use a name that represents the purpose of this specific inbound integration. For the **Interface** field, enter one of the following values.
 - For a message reply plugin, type `InboundIntegrationMessageReply`.
 - For a startable plugin for non-messaging use, type `InboundIntegrationStartablePlugin`.
 - c) Click the plus (+) symbol to add a plugin implementation and choose **Add Java plugin**.
 - d) In the **Java class** field, enter one of the following plugin types.
 - For a message reply plugin, type
`com.guidewire.pl.integration.inbound.file.DefaultFileInboundIntegrationMessageReply`.
 - For a startable plugin for non-messaging use, type
`com.guidewire.pl.integration.inbound.file.DefaultFileInboundIntegrationPlugin`.
 - e) Add a plugin parameter with the key `integrationservice`. For the value, type the unique name for your integration that you used in `inbound-integration-config.xml` for this integration.
6. Write your own inbound integration handler plugin implementation.
 - For a message reply plugin, your handler code must implement the plugin interface
`gw.plugin.integration.inbound.InboundIntegrationMessageReplyHandler`.
 - For a startable plugin for non-messaging use, your handler code must implement the plugin interface
`gw.plugin.integration.inbound.InboundIntegrationHandlerPlugin`.
- Both interfaces have one method called `process`, which has a single argument of type `Object`. The method has no return value. The file integration calls that method to process one message. Downcast this `Object` to `String` or `java.nio.file.Path` before using it.
7. Create the inbound file integration handler plugin to register your handler plugin implementation class.

- a) In Studio, in the Plugins registry, add a new .gwp file.
 - b) Set the interface name to the handler interface that you implemented. The Name field must match the <pluginhandler> subelement in the inbound-integration-config.xml file for this integration.
 - c) Click the plus (+) symbol to add a plugin implementation and choose the type of class that you implemented.
 - d) In the class name field, type or navigate to the class that you implemented.
8. Start the server in Guidewire Studio and test your new inbound integration by placing one or more files in the incoming directory.
- If inbound file integration does not succeed, review your antivirus settings on this server. Some antivirus applications interfere with file notifications that normal inbound file integration requires.

What to do next

See also

- “Inbound integration handlers for file and JMS integrations” on page 308
- “Inbound integration configuration XML elements” on page 311
- “Example of an inbound file integration” on page 318

Example of an inbound file integration

The following example configures inbound file integration in the inbound-integration-config.xml file.

```
<file-integration name="simpleFileIntegration" disabled="false">
  <threadpool>gw_default</threadpool>
  <pollinginterval>15</pollinginterval>
  <throttleinterval>5</throttleinterval>
  <ordered>true</ordered>
  <stoponerror>false</stoponerror>
  <transactional>false</transactional>
  <osgiservice>true</osgiservice>
  <traceabilityidcreationpoint>INBOUND_INTEGRATION_FILE</traceabilityidcreationpoint>
  <pluginhandler>SimpleInboundFileIntegrationHandler</pluginhandler>
  <processingmode>line</processingmode>
  <incoming>/tmp/file/incoming</incoming>
  <processing>/tmp/file/processing</processing>
  <error>/tmp/file/error</error>
  <done>/tmp/file/done</done>
  <charset>UTF-8</charset>
  <createdirectories>true</createdirectories>
</file-integration>
```

The following Gosu example implements a SimpleFileIntegrationHandler handler class to print the lines in the file.

```
package mycompany.integration
uses gw.plugin.integration.inbound.InboundIntegrationHandlerPlugin

class SimpleFileIntegrationHandler implements InboundIntegrationHandlerPlugin {
    // This example assumes that the inbound-integration-config.xml file
    // sets this integration to use "line" not "entire file" processing
    // See the <processingmode> element
    construct(){
        print("***** SimpleFileIntegration startup")
    }

    override function process(obj: Object) {
        // Downcast as needed (to String or java.nio.file.Path, depending on value of <processingmode>
        var line = obj as String
        print("***** SimpleFileIntegration processing one line of file: ${line} (!)")
    }
}
```

The example has two plugin implementations registered in the Plugins registry.

SimpleInboundFileIntegrationStartable.gwp

Registers an implementation of `InboundIntegrationStartablePlugin` with the Java class `com.guidewire.pl.integration.inbound.file.DefaultFileInboundIntegrationPlugin`. The `integrationService` plugin parameter is set to `simpleFileIntegration`.

SimpleInboundFileIntegrationHandler.gwp

Registers an implementation of `InboundIntegrationHandlerPlugin` with the Gosu class `mycompany.integration.SimpleFileIntegrationHandler`.

When you start up the server, the log line from starting up the file inbound integration handler appears in the console.

```
***** SimpleFileIntegration startup
```

The console also displays information about the directories that the inbound file integration uses for incoming, processing, and completed files. These directories are specified by the `<incoming>`, `<processing>`, and `<done>` elements in the `<file-integration>` definition. For the integration definition in this topic, the lines look like the following ones.

```
incoming [C:\tmp\file\incoming]
processing [C:\tmp\file\processing]
done [C:\tmp\file\done]
```

When you add files to the `/tmp/file/incoming` directory, you will see additional lines for each processed line. For example, add a file with the following content to the incoming directory:

```
Line 1
Line 2
Last line
```

The console contains lines that look like the following ones.

```
***** SimpleFileIntegration processing one line of file: Line 1 (!)
***** SimpleFileIntegration processing one line of file: Line 2 (!)
***** SimpleFileIntegration processing one line of file: Last line (!)
```

See also

- “Create an inbound file integration” on page 317

Inbound JMS integration

The base configuration of ClaimCenter includes a high-performance multi-threaded integration with inbound queues of Java Message Service (JMS) messages. ClaimCenter provides two classes, one for processing message replies, and one as a startable plugin. You cannot modify the code in the plugin implementation class in Studio, but you can use one or more instances of these integrations to work with your own file data. The inbound JMS integration supports application servers that implement the JSR-236 specification. These application servers currently include IBM WebSphere, JBoss, and Oracle Weblogic. You develop your own class that processes an individual message, and the inbound JMS framework handles message dispatch and thread management.

ClaimCenter logs any exception that the integration process method in your code throws. The next actions that ClaimCenter takes depend on the `stoponerror` and `ordered` configuration subelements.

- If both the `stoponerror` and `ordered` subelements have the value `true`, ClaimCenter stops processing on that queue until the plugin is restarted or the server is restarted.
- If the `stoponerror` subelement is `false`, or `stoponerror` is `true` and the `ordered` subelement is `false`, ClaimCenter executes roll-back logic for any processing that has occurred for this message. For a typical implementation of a JMS queue, the message is returned to the queue for retrying.

Be sure to catch any errors in your processing code and log any issues so that an administrator can identify and resolve the problem.

ClaimCenter can use JMS implementations on the application server but ClaimCenter does not include its own JMS implementation. Guidewire does not support inbound JMS integration on a Tomcat server. To provide inbound JMS integration to your ClaimCenter application, deploy ClaimCenter on one of the other supported application servers. For additional advice on setting up or configuring an inbound JMS integration with ClaimCenter, contact Guidewire Customer Support.

See also

- “Using the inbound integration polling and throttle intervals” on page 315

Create an inbound JMS integration

ClaimCenter uses inbound JMS integration to read information from messages that an external system creates. This integration can be either a message reply plugin or a startable plugin.

The base configuration of ClaimCenter provides implementations for the message reply plugin and the startable plugin for inbound file integration. You can use these plugins as a template for your own plugin. ClaimCenter also provides implementations for the message reply handler plugin and the startable handler plugin for inbound JMS integration. You can use these plugins as a template for your own handler plugin. You provide the handler class that implements your business logic.

Procedure

1. In the Project window, navigate to **configuration > config > integration**, and then open the `inbound-integration-config.xml` file.
2. Configure the thread pools.

You can use a thread pool element in the base configuration as a template.

3. In the list of integrations, create one `<integration>` element of type `<jms-integration>`.
4. Set configuration parameter subelements, ensuring that the order of subelements is correct.

The `inbound-integration-config.xsd` file specifies the correct sequence of the subelements.

- a) Set values for JNDI properties in the `<jndi-properties>` subelement and its `<jndi-property>` subelements.

Typically, you set values for the naming factory, connection factory, and queue name for your JMS configurations.

The following lines show an example configuration for an Apache Artemis broker.

```
<jndi-properties>
  <jndi-property>
    <key>java.naming.factory.initial</key>
    <value>org.apache.activemq.artemis.jndi.ActiveMQInitialContextFactory</value>
  </jndi-property>
  <jndi-property>
    <key>connectionFactory.ConnectionFactory</key>
    <value>tcp://localhost:61616</value>
  </jndi-property>
  <jndi-property>
    <key>queue.MyQueue</key>
    <value>MyQueue</value>
  </jndi-property>
</jndi-properties>
```

- b) Set values for `<connectionfactoryjndi>` and `<destinationjndi>` subelements.

These values must match the JNDI property values for the connection factory and queue name.

For the example JNDI properties, these elements look like the following lines.

```
<connectionfactoryjndi>ConnectionFactory</connectionfactoryjndi>
<destinationjndi>MyQueue</destinationjndi>
```

5. Create the inbound JMS integration plugin.

- a) In Studio, in the Plugins registry, add a new .gwp file.
 - b) Studio prompts for a plugin name and plugin interface. For the plugin name, use a name that represents the purpose of this specific inbound integration. For the **Interface** field, enter one of the following values.
 - For a message reply plugin, type `InboundIntegrationMessageReply`.
 - For a startable plugin for non-messaging use, type `InboundIntegrationStartablePlugin`.
 - c) Click the plus (+) symbol to add a plugin implementation and choose **Add Java plugin**.
 - d) In the **Java class** field, enter one of the following plugin types.
 - For a message reply plugin, type
`com.guidewire.pl.integration.inbound.jms.DefaultJMSInboundIntegrationMessageReply`.
 - For a startable plugin for non-messaging use, type
`com.guidewire.pl.integration.inbound.jms.DefaultJMSInboundIntegrationPlugin`.
- e) Add a plugin parameter with the key `integrationservice`. For the value, type the unique name for your integration that you used in `inbound-integration-config.xml` for this integration.
6. Write your own inbound integration handler plugin implementation.
 - For a message reply plugin, implement the interface
`gw.plugin.integration.inbound.InboundIntegrationMessageReplyHandler`.
 - For a startable plugin for non-messaging use, implement the interface
`gw.plugin.integration.inbound.InboundIntegrationHandlerPlugin`.
- Both interfaces have one method called `process`, which has a single argument of type `Object`. The method has no return value. The JMS integration calls that method to process one message. Downcast this `Object` to `javax.jms.Message` before using it.
7. Create the inbound JMS integration handler plugin to register your handler plugin implementation class.
 - a) In Studio, in the Plugins registry, add a new .gwp file.
 - b) Set the interface name to the handler interface that you implemented.

The **Name** field must match the `<pluginhandler>` subelement in the `inbound-integration-config.xml` file for this integration.
 - c) Click the plus (+) symbol to add a plugin implementation and choose the type of class that you implemented.
 - d) In the class name field, type or navigate to the class that you implemented.
8. Start the server in Guidewire Studio and test your new inbound integration by sending one or more JMS messages to the queue that your class handles.

What to do next

See also

- “Inbound integration handlers for file and JMS integrations” on page 308
- “Inbound integration configuration XML elements” on page 311
- “Example of an inbound JMS integration” on page 321

Example of an inbound JMS integration

The following example configures inbound JMS integration in the `inbound-integration-config.xml` file. You can test the inbound JMS integration in dev mode on the Jetty platform, by using an unmanaged thread pool.

```
<jms-integration name="simpleJmsIntegration" disabled="false">
  <threadpool>gw_default</threadpool>
  <pollinginterval>15</pollinginterval>
  <throttleinterval>5</throttleinterval>
  <ordered>true</ordered>
```

```

<stoponerror>false</stoponerror>
<transactional>true</transactional>
<osgiservice>true</osgiservice>
<traceabilityidcreationpoint>INBOUND_INTEGRATION_JMS</traceabilityidcreationpoint>
<pluginhandler>SimpleInboundJmsIntegrationHandler</pluginhandler>
<connectionfactoryjndi>ConnectionFactory</connectionfactoryjndi>
<destinationjndi>MyQueue</destinationjndi>
<user/>
<password/>
<durablesubscription/>
<nolocal/>
<messageselector/>
<messagereceivetimeout>5</messagereceivetimeout>
<batchlimit>5</batchlimit>
<jndi-properties>
  <jndi-property>
    <key>java.naming.factory.initial</key>
    <value>org.apache.activemq.artemis.jndi.ActiveMQInitialContextFactory</value>
  </jndi-property>
  <jndi-property>
    <key>connectionFactory.ConnectionFactory</key>
    <value>tcp://localhost:61616</value>
  </jndi-property>
  <jndi-property>
    <key>queue.MyQueue</key>
    <value>MyQueue</value>
  </jndi-property>
</jndi-properties>
</jms-integration>

```

The following Gosu example implements a `SimpleJmsIntegrationHandler` handler class to print a line for each message.

```

package mycompany.integration

uses gw.plugin.integration.inbound.InboundIntegrationHandlerPlugin

class SimpleJmsIntegrationHandler implements InboundIntegrationHandlerPlugin {

  construct(){
    print("***** SimpleJmsIntegration startup")
  }

  override function process(obj: Object) {
    // Downcast to Message
    var msg = obj as javax.jms.Message
    print("***** SimpleJmsIntegration processing one message: ${msg} (!)")
  }
}

```

The example has two plugin implementations registered in the Plugins registry.

`SimpleInboundJmsIntegrationStartable.gwp`

Registers an implementation of `InboundIntegrationStartablePlugin` with the Java class `com.guidewire.pl.integration.inbound.file.DefaultJMSInboundIntegrationPlugin`. The `integrationservice` plugin parameter is set to `simpleJmsIntegration`.

`SimpleInboundJmsIntegrationHandler.gwp`

Registers an implementation of `InboundIntegrationHandlerPlugin` with the Gosu class `mycompany.integration.SimpleJmsIntegrationHandler`.

When you start up the server, the log line from starting up the inbound JMS integration handler appears in the console.

```
***** SimpleJmsIntegration startup
```

When you send messages to the JMS queue, you will see additional lines for each processed message. For example, the console contains lines that look like the following ones.

```

***** SimpleJmsIntegration processing one message: ActiveMQMessage[null]:PERSISTENT/
ClientMessageImpl[messageID=88,
durable=true,address=jms.queue.MyQueue(userID=null,properties=TypedProperties[__HDR_BROKER_IN_TIME=1536106159084,__HDR_AR
RIVAL=0,
__HDR_GROUP_SEQUENCE=0,__HDR_COMMAND_ID=7,__HDR_PRODUCER_ID=[...],
__HDR_MESSAGE_ID=[...],__HDR_DROPPABLE=false]] (!)
***** SimpleJmsIntegration processing one message: ... (!)
***** SimpleJmsIntegration processing one message: ... (!)

```

See also

- “Create an inbound JMS integration” on page 320

Custom inbound integrations

The base configuration of ClaimCenter includes inbound integrations of file-based input and JMS messages. If these integrations do not serve your needs, you can write your own inbound integration. The major component of the integration is a work agent, which is a service that coordinates and processes work. The work agent interface, `gw.api.integration.inbound.WorkAgent`, defines the core behavior of the inbound integration framework. Writing a work agent is the most complex part of writing your own custom inbound integration. To support the work agent, you write additional classes based on interfaces in the `gw.api.integration.inbound` package hierarchy. Additionally, you write a plugin that starts and stops the work agent.

- For a startable plugin for non-messaging contexts, implement the `gw.api.startable.IStartablePlugin` interface. This interface defines the methods `start`, `stop`, and `getState`.
- For a message reply plugin, implement the `gw.plugin.messaging.MessageReply` interface. This interface supports classes that handle replies from messages that a `MessageTransport` implementation sends.

After you write your custom implementation of an inbound integration, use the Guidewire Studio Plugins Registry to register your plugin implementation with ClaimCenter.

Writing a custom inbound integration plugin

Depending on your needs, you write a startable plugin or a message reply plugin. The plugin is the controller that starts and stops the work agent.

Writing a startable plugin for a custom inbound integration

For a startable plugin, implement the `gw.api.startable.IStartablePlugin` interface.

Start the work agent in the `start` method. Stop the work agent in the `stop` method.

- In the `start` method, call `CustomWorkAgent.startCustomWorkAgent(integrationName)`.
- In the `stop` method, call `CustomWorkAgent.stopCustomWorkAgent(integrationName)`.

For the argument to those `gw.api.integration.inbound.CustomWorkAgent` methods, pass the inbound integration name. This name is the `name` attribute on the `<custom-integration>` element in your `inbound-integration-config.xml` file.

The following Java example demonstrates this pattern for the controller of a custom implementation of a startable plugin.

```
import gw.api.integration.inbound.CustomWorkAgent;
import gw.api.server.Availability;
import gw.api.server.AvailabilityLevel;
import gw.api.startable.IStartablePlugin;
import gw.api.startable.StartablePluginCallbackHandler;
import gw.api.startable.StartablePluginState;

@Availability(AvailabilityLevel.MULTIUSER)
public class CustomStartableInboundIntegrationController implements IStartablePlugin {
    private String _name = "exampleCustomIntegration";

    public CustomStartableInboundIntegrationController() {
        ...
    }

    private void start(StartablePluginCallbackHandler pluginCallbackHandler, boolean serverStarting) {
        try {
            CustomWorkAgent.startCustomWorkAgent(_name);
        } catch (GWLifecycleException e) {
            throw new RuntimeException(e);
        }
    }

    private void stop(boolean serverShuttingDown) {
        try {
```

```

        CustomWorkAgent.stopCustomWorkAgent(_name);
    } catch (GWLifecycleException e) {
        throw new RuntimeException(e);
    }
}

...
}

```

Writing a message reply plugin for a custom inbound integration

For a message reply plugin, implement the `gw.plugin.messaging.MessageReply` interface.

Start the work agent in the `initTools` method. Stop the work agent in the `shutdown` method.

- In the `initTools` method, call `CustomWorkAgent.startCustomWorkAgent(integrationName)`.
- In the `shutdown` method, call `CustomWorkAgent.stopCustomWorkAgent(integrationName)`.

For the argument to those `gw.api.integration.inbound.CustomWorkAgent` methods, pass the inbound integration name. This name is the `name` attribute on the `<custom-integration>` element in your `inbound-integration-config.xml` file.

The following Java example demonstrates this pattern for the controller of a custom implementation of a message reply plugin.

```

import gw.api.integration.inbound.CustomWorkAgent;
import gw.plugin.PluginCallbackHandler;
import gw.plugin.messaging.MessageFinder;
import gw.plugin.messaging.MessageReply;

@Availability(AvailabilityLevel.MULTIUSER)
public class CustomMessagingInboundIntegrationController implements MessageReply {

    private String _name = "exampleCustomIntegration"

    public CustomMessagingInboundIntegrationController() {
        ...
    }

    private void initTools(PluginCallbackHandler handler, MessageFinder msgFinder) {
        try {
            CustomWorkAgent.startCustomWorkAgent(_name);
        } catch (GWLifecycleException e) {
            throw new RuntimeException(e);
        }
        ...
    }

    private void shutdown() {
        try {
            CustomWorkAgent.stopCustomWorkAgent(_name);
        } catch (GWLifecycleException e) {
            throw new RuntimeException(e);
        }
        ...
    }

    ...
}

```

See also

- “[Startable plugins](#)” on page 299

Writing a work agent implementation

The `gw.api.integration.inbound.WorkAgent` interface defines methods that coordinate and process work. You must write your own class that implements this interface.

To write a complete work agent implementation, you must write multiple related classes that work together. The following tables provide a summary of each class.

The following class is your top-level class.

Class that you write	Interface to implement	Description
A work agent	WorkAgent in package gw.api.integration.inbound.	The work agent implementation is the top level class that coordinates work for this service. The work agent instantiates your class that implements the interface Factory.

You use the following classes to find and prepare work during each polling interval.

Class that you write	Interface to implement	Description
A factory	Factory in package gw.api.integration.inbound.	For each polling interval, the factory instantiates the class that implements the interface WorkSetProcessor.
A work set processor	WorkSetProcessor in package gw.api.integration.inbound.work. If your plugin supports the optional feature of being transactional, implement the subinterface TransactionalWorkSetProcessor.	The work set processor acquires and allocates resources. This class also instantiates the class that implements the interface Inbound. The main method for processing one unit of work in a WorkData object is the process method in this class.
A class to find work	Inbound in package gw.api.integration.inbound.work.	The inbound class defines how to find work in its findWork method that returns new work data sets. This class also instantiates the class that implements the interface WorkDataSet.

The following classes represent the work itself.

Class that you write	Interface to implement	Description
A work data set	WorkDataSet in package gw.api.integration.inbound.work.	An object of this class represents the set of all data that the Inbound object found in this polling interval. This class encapsulates a set of work data (WorkData) objects and any necessary context information to operate on the data. This WorkDataSet object also instantiates the class that implements the WorkData interface.
Work data	WorkData in package gw.api.integration.inbound.work.	An object of this class represents one unit of work.

Setting up and tearing down the work agent

In your WorkAgent implementation class, write a `setup` method that initializes your resources. ClaimCenter calls the `setup` method before the `start` method.

```
public void setup(Map<String, Object> properties);
```

The `java.util.Map` object that is the method argument is the set of plugin parameters from the Studio Plugins Editor for your plugin implementation.

Implement the `teardown` method to release resources acquired in the `start` method. ClaimCenter calls the `teardown` method before the `stop` method.

Starting and stopping the work agent

In your WorkAgent implementation class, implement the plugin method `start` to start the work listener and perform any necessary initialization that must happen each time you start the work agent.

Implement the plugin method `stop` and perform any necessary logic to stop your work agent.

Compare and contrast the `start` and `stop` methods with the `setup` and `teardown` methods.

If you use the `InboundIntegrationStartablePlugin` startable plugin variant in the inbound integration plugin, be aware that the main `WorkAgent` interface defines `start` and `stop` methods with no arguments. The startable plugin variant implements the interface `IStartablePlugin`. This interface defines method signatures of the `start` and `stop` methods that take arguments. These `start` and `stop` methods must call the appropriate method of the utility class `gw.api.integration.inbound.CustomWorkAgent`.

- In each `start` method, call `CustomWorkAgent.startCustomWorkAgent(integrationName)`.
- In each `stop` method, call `CustomWorkAgent.stopCustomWorkAgent(integrationName)`.

For the argument to those `gw.api.integration.inbound.CustomWorkAgent` methods, pass the inbound integration name in the `name` attribute on the `<custom-integration>` element in your `inbound-integration-config.xml` file.

The following Java example demonstrates this pattern for the controller of a custom implementation of a startable plugin.

```
import gw.api.integration.inbound.CustomWorkAgent;
import gw.api.server.Availability;
import gw.api.server.AvailabilityLevel;
import gw.api.startable.IStartablePlugin;
import gw.api.startable.StartablePluginCallbackHandler;
import gw.api.startable.StartablePluginState;

@Availability(AvailabilityLevel.MULTIUSER)
public class CustomStartableInboundIntegrationController implements IStartablePlugin {
    private String _name = "exampleCustomIntegration";

    ...

    private void start( StartablePluginCallbackHandler pluginCallbackHandler, boolean serverStarting ) {
        try {
            CustomWorkAgent.startCustomWorkAgent( _name );
        } catch ( GWLifecycleException e ) {
            throw new RuntimeException( e );
        }
    }

    private void stop( boolean serverShuttingDown ) {
        try {
            CustomWorkAgent.stopCustomWorkAgent( _name );
        } catch ( GWLifecycleException e ) {
            throw new RuntimeException( e );
        }
    }
}
```

Declaring whether your work agent is transactional

ClaimCenter determines whether a work agent is transactional by calling the `transactional` plugin method. A transactional work agent creates work items with a slightly different interface. There are additional methods that you must implement to begin work, commit work, and roll back transactional changes to partially finished work. To specify that your work agent is transactional, return `true` from the `transactional` method. Otherwise, return `false`.

If your work agent is transactional, your implementation of the `Factory.createWorkUnit` method must return an instance of `TransactionalWorkSetProcessor` instead of `WorkSetProcessor`. The `TransactionalWorkSetProcessor` interface extends `WorkSetProcessor`.

Getting a factory for the work agent

In your `WorkAgent` implementation class, implement the `factory` method. This method must return a `Factory` object, which represents an object that creates work data sets.

The `Factory` interface defines a single method called `createWorkProcessor`. The method takes no arguments and returns an instance of your own custom class that implements the `WorkSetProcessor` interface.

If your agent is transactional, your implementation of the `Factory.createWorkProcessor` method must return an instance of `TransactionalWorkSetProcessor` instead of `WorkSetProcessor`. The `TransactionalWorkSetProcessor` interface extends `WorkSetProcessor`. Both interfaces are in the `gw.api.integration.inbound.work` package.

Writing a work set processor

To perform your actual work, you create a class called a work set processor, which implements the `WorkSetProcessor` interface or its subinterface `TransactionalWorkSetProcessor`. Both interfaces are in the `gw.api.integration.inbound.work` package.

The basic `WorkSetProcessor` interface defines two methods.

getInbound

Gets an object that acquires and divides resources to create work items. This method returns an object of type `Inbound`, which is an interface that defines a single method called `findWork`. Define your own class that implements the `Inbound` interface. The `findWork` method must get the work data set, which represents multiple work items. If your plugin supports unordered multi-threaded work, each work item represents work that can be done by its own thread. For example, for the inbound file integration, a work data set is a list of newly added files. Each file is a separate work item. The `findWork` method returns the data set encapsulated in a `WorkDataSet` object. The polling process of the inbound integration framework calls the `findWork` method to do the main work of getting new data to process. From Gosu, this method appears as a getter for the `Inbound` property rather than as a method.

process

Processes one work data item within a work data set. The method takes two arguments of type `WorkContext` and `WorkData`. The `WorkData` argument is one work item in the work data set. You can optionally choose to use the `WorkContext` argument to declare a resource or other context necessary to process the data item. Your implementation of `WorkDataSet` populates this context information if you need it. For example, if your inbound integration is listening to a message queue, you might store the connection or queue information in the `WorkContext` object. Your `WorkSetProcessor` can then access this connection information in the `process` method when processing one message on the queue.

If your agent is transactional, your implementation of the `Factory.createWorkUnit` method must return an instance of `TransactionalWorkSetProcessor` instead of `WorkSetProcessor`. The `TransactionalWorkSetProcessor` interface extends `WorkSetProcessor`.

The `TransactionalWorkSetProcessor` interface defines several additional methods.

begin

Begin any necessary transactional context. You are responsible for management of any transactions.

commit

Commit any changes. You are responsible for management of any transactions.

rollback

Rollback any changes. You are responsible for management of any transactions.

All three methods take a single argument of type `TxContext`. The `TxContext` interface extends `WorkContext`. Use the `TxContext` object to represent work context information that also contains transaction-specific information. Create your own implementation of this class in your `getContext` method of your `WorkDataSet`.

Error handling for a custom inbound integration

If you throw an exception in your code, ClaimCenter behavior depends on the `stoponerror` and `ordered` configuration parameters. If `stoponerror` is `true`, in the following cases, ClaimCenter immediately stops processing until the plugin is restarted or the server is restarted:

- The `WorkDataSet.findWork` method throws any exception.
- The `ordered` configuration parameter is also `true` and the `WorkDataSet.process` method throws any exception.

If the `ordered` configuration parameter is `false`, ClaimCenter logs any exception that the `WorkSetProcessor` object's `process` method throws and the item is skipped. It is important to catch any exceptions in the `process` method to ensure that you correctly handle error conditions. For example, you might need to notify administrators or place the work item in a special location for appropriate handling.

See also

- “Using the inbound integration polling and throttle intervals” on page 315

Creating a work data set

You must implement your own class that encapsulates knowledge about a work data set, which represents the set of all data found in this polling interval. The work data set is created by your implementation of the `Inbound.findWork` method. For example, an inbound file integration creates a work data set representing a list of all new files in an incoming directory.

Create a class that implements the `gw.api.integration.inbound.work.WorkDataSet` interface. Your class must implement the following methods.

`getData`

Get the next work item and move any iterator that you maintain forward one item so that the next call returns the next item after this one. Return a `WorkData` object if there are more items to process. Return `null` to indicate no more items. For example, an inbound file integration might return the next item in a list of files. The `WorkData` interface is a marker interface, so it has no methods. Write your own implementation of a class that implements this interface. Add any object variables necessary to store information to represent one work item. It is the `WorkDataSet.getData` method that is responsible for instantiating the appropriate class that you write and populating any appropriate data fields. For example, for an inbound file integration, one `WorkData` item might represent one new file to process.

In a Gosu implementation, the `getData` method appears as a getter for the `Data` property, not a method.

`hasNext`

Return `true` if there are any unprocessed items, otherwise return `false`. In other words, if this same object’s `getData` method would return a non-null value if called immediately, return `true`.

`getContext`

Your implementation of a work data set can optionally declare a resource or other context necessary to process the data item. You are responsible for populating this context information if you need it. For example, if your inbound integration listens to a message queue, store the connection or queue information in an instance of a class that you write that implements `gw.api.integration.inbound.WorkContext`. In your `WorkSetProcessor` implementation, you can access this connection information from the `WorkSetProcessor` object’s `process` method when processing each new message.

In a Gosu implementation, the `getContext` method appears as a getter for the `Context` property, not a method.

`close`

Close and release any resources acquired by your work data set.

If your plugin supports transactional work items, your class must implement the interface `gw.api.integration.inbound.work.TxContext`, which requires two additional methods.

`isRollback`

Returns a boolean value that indicates the transaction will be rolled back.

In a Gosu implementation, the `isRollback` method appears as a getter for the `Rollback` property, not a method.

`setRollbackOnly`

Set your own boolean value to indicate that a rollback will occur.

Getting parameters for a work agent plugin implementation

Like all other plugin types, your plugin implementation can get parameter values. The `Map` argument to the `setup` method includes all parameters that you set in the `inbound-integration-config.xml` file for that integration. Save the map or the values of important parameters in private variables in your plugin implementation.

The Map argument to the `setup` method also includes any arbitrary parameters that you set in the `<parameters>` configuration element.

Install a custom inbound integration

ClaimCenter uses a custom inbound integration to read information that an external system creates and that is neither a file nor a JMS message. This integration can be either a message reply plugin or a startable plugin.

Procedure

1. In the **Project** window, navigate to **configuration > config > integration**, and open the file `inbound-integration-config.xml`.
2. Configure the thread pools.
You can use a thread pool element in the base configuration as a template.
3. In the list of integrations, create one `<integration>` element of type `<custom-integration>`.
You can use a `<custom-integration>` element in the base configuration as a template.
4. Set configuration parameter subelements.
5. Create the custom inbound integration plugin.
 - a) In Studio, in the Plugins registry, add a new `.gwp` file.
 - b) Studio prompts for a plugin name and plugin interface. For the plugin name, use a name that represents the purpose of this specific inbound integration. For the **Interface** field, enter one of the following values.
 - For a message reply plugin, type `InboundIntegrationMessageReply`.
 - For a startable plugin for non-messaging use, type `InboundIntegrationStartablePlugin`.
 - c) Click the plus (+) symbol to add a plugin implementation and choose the class type that you implemented.
 - d) For a Gosu or Java plugin, in the **Gosu class** or **Java class** field, type the fully qualified name of your plugin implementation class. For an OSGi plugin, in the **Service PID** field, type the fully qualified Java class name of your OSGi implementation class.
 - e) Add a plugin parameter with the key `integrationservice`. For the value, type the unique name for your integration that you used in `inbound-integration-config.xml` for this integration.
6. Start the server and test your new inbound integration. Add logging code as appropriate to confirm the integration.

What to do next

See also

- “Inbound integration configuration XML elements” on page 311

Redis integration

In the base configuration, Guidewire provides an implementation of the `ClusterBroadcastTransportFactory` plugin that provides a mechanism for communications between the members of a Guidewire ClaimCenter cluster. The default version of this plugin implements OSGi class `DefaultDatabaseBroadcastTransport`. In the base configuration, ClaimCenter uses this mechanism for default message broadcast if you do not enable the `ClusterFastBroadcastTransportFactory` plugin implementation.

However, it is possible to implement an optional version of the `ClusterBroadcastTransportFactory` that implements OSGi class `RedisBroadcastTransportFactory`. Use of this plugin version requires additional integration and configuration steps.

Using Redis with a Guidewire cluster

In general practice, Guidewire recommends that you set up three redundant Redis servers (without Sentinel) on different physical hosts to avoid a single point of failure. Even in the unlikely event that all Redis servers failed simultaneously, the ClaimCenter cluster will continue to work. Only broadcast messages (including cache evictions) will stop working with all the Redis servers down, which can cause a rise in the frequency of CDCEs (Concurrent Data Change Exceptions) and other potential problems. However, as long as at least one of the redundant Redis servers is working and reachable by all cluster nodes, Guidewire applications will continue working with very minimal impact.

Guidewire recommendations

Guidewire makes the following general recommendations in working with a Redis server.

Redis server

Ensure that you turn off the Redis persistent settings.

Set the following value in addition to the Redis default values:

```
client-output-buffer-limit pubsub 2000mb 1000mb 60
```

RedisBroadcastTransportFactory plugin

Set the `batchWriteInterval` plugin parameter to be between 100 to 200 (ms). Start at the upper value and gradually reduce the value until the number of CDCEs approaches zero.

Application log

Monitor the log entries closely during testing. Multiple occurrences of the following log entry indicate that a Redis server is down:

```
redis.clients.jedis.exceptions.JedisConnectionException
```

Set up Guidewire cluster with Redis server

It is possible to configure a Redis server to work with Guidewire InsuranceSuite application clusters.

About this task

To run a Guidewire ClaimCenter cluster with a Redis server, you need to perform the following tasks:

- Configure the Redis server with specific integration parameters.
- Update the `BroadcastTransportFactory` plugin to support Redis.
- Redeploy the ClaimCenter application server.

IMPORTANT: Guidewire does not support the Redis Sentinel configuration. See the Product Support Matrix for supported configurations.

Procedure

Configure the Redis server

1. Download Redis from the following location to your local file directory:

<https://redis.io/download>

2. Install the Redis distribution using the following commands. Enter the appropriate version numbers for your Redis distribution download.

```
$ tar xzf redis-5.0.7.tar.gz
$ cd redis-5.0.7
$ make
```

3. Open file `/etc/redis/redis.conf` and set the following parameters:

```
# Run Redis as background process
daemonize yes

# Turn off protected mode
protected-mode no

# Set Redis port value
port 6379

# Specify the value (in milliseconds) above which Redis logs latency data
latency-monitor-threshold 10

# Set buffer size (for slow clients)
client-output-buffer-limit pubsub 2000mb 1000mb 60

# Define Redis logging parameters
loglevel notice
logfile "./redis.log"
```

4. Use the following command to start the Redis server with the updated Redis configuration file:

`src/redis-server ./redis.conf`

Create the Guidewire Redis transport plugin

5. In Guidewire Studio for ClaimCenter, navigate to the following location in the Studio **Project** window:

configuration > config > Plugins > registry

6. Open plugin `ClusterBroadcastTransportFactory` for editing.

- a) Select the current OSGi entry in the plugin editor, remove its **Service PID** value, and enter the following text in its place:

`com.guidewire.pl.cluster.internal.broadcast.redis.RedisBroadcastTransportFactory`

- b) Add host URL and port parameters for each Redis server that you are deploying.

- c) Add a `maxBatchWriteInterval` parameter and set its value to 100 (one second).

After you complete this work, you see an entry similar to the following for the updated OSGi implementation:

```
<plugin-osgi servicepid="com.guidewire.pl.cluster.internal.broadcast.redis.RedisBroadcastTransportFactory">
  <param name="redisServer1" value="Redis server host name or IP"/>
  <param name="redisPort1" value="6379"/>
  <param name="redisServer2" value="Redis server host name or IP"/>
  <param name="redisPort2" value="6279"/>
  ...
  <param name="maxBatchWriteInterval" value="100"/>
</plugin-osgi>
```

Redeploy and restart the ClaimCenter application

7. Recompile the changed resource using one of the following methods:
 - Select **Build Project** or **Rebuild Project** from the Studio **Build** menu.
 - Execute `gwb compile` from a server command prompt (after you redeploy to that server).
8. Redeploy the ClaimCenter application to each server in the cluster.
9. Stop and restart the application server to update the ClaimCenter application database.

After restarting the application, you see an entry similar to the following in the ClaimCenter log:

```
2020-2-17 09:42:44,744 INFO [main] Server.Cluster.RedisBroadcastTransport
RedisPublisher(someHost.myCompany.com:6379) connected to Redis
```

Example Redis setup

Guidewire recommends that your Redis cluster configuration include several redundant Redis servers. You differentiate between the multiple servers through the use of plugin parameters in your Redis implementation of the `ClusterBroadcastTransport` plugin. The following example plugin code illustrates how to create a plugin that defines two separate Redis servers.

```
<?xml version="1.0"?>
<plugin interface="ClusterBroadcastTransportFactory" name="ClusterBroadcastTransportFactory">
  <plugin-osgi servicepid="com.guidewire.pl.cluster.internal.broadcast.redis.RedisBroadcastTransportFactory">
    <param name="redisServer1" value="localhost"/>
    <param name="redisPort1" value="6379"/>
    <param name="redisServer2" value="localhost"/>
    <param name="redisPort2" value="6279"/>
  </plugin-osgi>
```

Note: In actual practice, Guidewire recommends that you use different physical hosts for each Redis server.

To create this plugin configuration, first navigate to the following location in the ClaimCenter Studio **Project** window:

configuration > config > Plugins > registry > ClusterBroadcastTransportFactory

In the plugin editor, click the plus icon (+) underneath **Parameters** to add the necessary plugin parameter names and values.

Redis plugin parameters

In the base configuration, Guidewire provides the following OSGi implementations classes for plugins that implement the `ClusterBroadcastTransportFactory` interface.

`DefaultDatabaseBroadcastTransport` OSGi implementation class that uses the application database for broadcasting messages. The broadcast mechanism works by writing messages to a database table with the plugin periodically loading recently added message rows to this table.
ClaimCenter uses this mechanism for default message broadcast if you do not enable the `ClusterFastBroadcastTransportFactory` plugin implementation.

RedisBroadcastTransportFactory	OSGi implementation class that uses Redis in broadcasting messages within a Guidewire cluster. This is an optional plugin implementation that you must configure (and then enable) to use.
---------------------------------------	--

The two plugin implementations share multiple parameters, and additionally, each implementation provides its own parameters for configuring the plugin. Use the configuration parameters to provide precise control over the behavior of plugins that implement the `ClusterBroadcastTransportFactory` interface.

RedisBroadcastTransportFactory parameters

The only required Redis plugin parameter is `redisServer`.

channel

Name of the Redis channel to which the transport is to publish messages. The default is `GW_AppCode`, with `AppCode` being the (capitalized) two-letter acronym for the Guidewire InsuranceSuite application running in the cluster.

maxTimeDiffBetweenServers

If the cluster contains multiple Redis servers, and there is a gap in the message stream from one server, the transport waits for other servers to fill the gap. However, if the other servers are too slow and the time difference between the newest messages (each message has timestamp inside) from the servers is greater than this value, the transport stops waiting for missing messages and moves forward. The default is 30000 (30 seconds).

redisPassword

Password of the Redis server. The default value is no password. If you have several Redis servers, create a sequence of Redis password names. For example, create the following sequence: `redisPassword1`, `redisPassword2`, ..., `redisPasswordN`.

redisPort

Optional port of the Redis server. The default port value is 6379. If you have several Redis servers, create a sequence of Redis port names. For example, create the following sequence: `redisPort1`, `redisPort2`, ..., `redisPortN`.

redisServer

IP address or host name of Redis server. If you have several Redis servers use, create a sequence of Redis server names. For example, create the following sequence: `redisServer1`, `redisServer2`, ..., `redisServerN`. You must supply a value for this parameter.

socketTimeout

TCP socket timeout (in milliseconds) to use while establishing connection to Redis. The default is 60000 (1 minute).

Common parameters

batchWriteAttempts

Maximum number of attempts to write to a batch queue. If the number of consecutive errors exceeds this threshold, the transport switches to ERROR mode in which each new messages pops the oldest message out of the in-memory queue. The purpose of this parameter value is to avoid out-of-memory issues. The default is 10.

batchWriteInterval

Maximum time interval to wait (in milliseconds) before ClaimCenter writes, or sends, the current batch of messages. This parameter has the following default:

- `DefaultDatabaseBroadcastTransportFactory` - 2000 (2 seconds)
- `RedisBroadcastTransportFactory` - 1000 (1 second)

maxBatchWriteInterval

Maximum amount (in milliseconds) to which the retry interval can grow in making multiple attempts to resend a failed message batch. If the broadcast transport generates an error while trying to send the current batch of messages, the plugin attempts to send the message batch with an exponential backoff. The exponential backoff

doubles the delay after each consecutive error up to the stipulated value, which is the maximum retry interval. This parameter has the following default:

- `DefaultDatabaseBroadcastTransportFactory` - 16000 (16 seconds)
- `RedisBroadcastTransportFactory` - 8000 (8 seconds)

maxOutboundBufferSize

Maximum size of outbound buffer (in megabytes). The purpose of this parameter value is to prevent out-of-memory issues if a transport is having problems writing or sending messages. The default is 25 megabytes.

preferredBatchDataSize

Maximum size of the batch (in bytes). If the size of the current batch (the sum of all of the message batch sizes) reaches this threshold, ClaimCenter writes, or sends, the current batch of messages immediately. This value must be less than or equal to the largest possible integer value supported by your hardware.

preferredBatchMessageCnt

Maximum number of pending messages allowed in the batch queue. If the number of messages in the batch queue reaches this threshold, ClaimCenter writes, or sends, the current batch of messages immediately. The value must be less than or equal to the largest possible integer value supported by your hardware.

receiverPoolSize

Number of threads in the thread pool that handle inbound messages. The default is 4.

Accessing Redis information

There are several different ways that you can access information about your Redis configuration and server performance:

- Access the Server Tools **Management Beans** screen
- Review the application logs

Management Bean information

Guidewire provides information about the Redis configuration (and other application configurations) in the Server Tools **Management Beans** screen. To access this screen, log into Guidewire ClaimCenter using an administrative account that contains the `soapadmin` permission.

In the Management Beans screen, click the following link:

`gw.plugin.cluster.RedisBroadcastTransport:type=broadcast`

In the **Guidewire Managed Bean Properties** screen that opens, you see information related to each defined Redis server.

Application log entries

The ClaimCenter application logs provide information on the installed Redis servers, for example:

```
main 12:06:37,360 INFO Server.Cluster.RedisBroadcastTransport RedisPublisher(ph-app-s011.guidewire.com:6379) connected to Redis
```

You can also set up additional cluster logging using Guidewire intentional logging (business event logging). To do so, you must set up the following logging category:

`SERVER_CLUSTER`

See the *Integration Guide* for more information.

part 4

Messaging

You can send messages to external systems after something changes in ClaimCenter, such as a changed exposure or an added check. The changes trigger events, which trigger your code that sends messages to external systems. For example, if you create a new check in ClaimCenter, your messaging code notifies a corporate financials system or notifies a check printing service.

ClaimCenter defines a large number of events of potential interest to external systems. In response to events of interest, rules can be written to generate messages intended for external systems. ClaimCenter queues the messages and then dispatches them to the appropriate external systems.

This topic explains how ClaimCenter generates messages in response to events and how to connect external systems to receive those messages.

Messaging and events

To work with messaging and the events that generate messages, you need to be familiar with the following concepts.

Event

An event is an abstract notification of a change in ClaimCenter that might be interesting to an external system. An event most often represents a user action to application data, or an API that changed application data. Entity types defined with the `<events>` tag trigger an event if a user action or API adds, changes, or removes data associated with an instance of the entity. Each event is identified by a `String` name.

For example, in ClaimCenter, adding a new check on a claim triggers an event. The event name is "CheckAdded".

A single user action or API call might trigger multiple events for different objects in a single database transaction.

When an event is triggered, a call is made to the Event Fired rule set for each message destination registered to process the event. The rule set performs the operations necessary to process the event, which can include sending a message to an external system.

Message

A message is information to send to an external system in response to an event. Messages are created while executing the Event Fired rule set that is called in response to the triggering of a particular event. ClaimCenter can ignore the event or send one or more messages to each external system that cares about that event. Each message includes a message payload that contains the content of the message. The message payload is a `String` stored in the `Message.Payload` property.

Message history

After a message is sent, the application converts a `Message` object to a `MessageHistory` object. `MessageHistory` objects are saved in a database table. The database table can be used to detect duplicate messages and also to track and understand the messaging history of external systems.

Messaging destinations

A messaging destination is an external system that receives messages from ClaimCenter.

A messaging destination typically represents a single external system. Register a destination once for each external system, even if the system is used for multiple types of data.

Messaging destinations are registered in Guidewire Studio.

Each destination can register a list of event names for which it needs to receive notifications. The Event Fired rule set runs once for every combination of an event and a destination interested in that event. An Event Fired rule can

determine the destination for a particular triggered event by examining the `DestinationID` property of the message context object.

Root object

A root object for an event is the entity instance that is most associated with the event. This might be the primary entity that is the top of a hierarchy of objects, or it might a small subobject.

Separate from the concept of a root object for an event, each message has a root object. By default, the message's root object is the same as the root object for the event that triggered the Event Fired rules within which you created the message. This default makes sense in most cases. You can override this default for a message if necessary.

Primary entity and primary object

A primary entity represents a type of high-level object that a Guidewire application uses to group and sort related messages. A primary object is a specific instance of a primary entity. Each Guidewire application specifies a default primary entity type for the application, or no default primary entity type.

Additionally, messaging destinations can override the primary entity type, and that setting applies just to that messaging destination. Only a subset of entity types are supported as primary entities for each Guidewire application.

Determining which primary object, if any, applies for a messaging destination is critical to understanding how ClaimCenter orders messages.

- The default primary entity is `Claim`. Most objects are subobjects of a claim. An `Exposure` object is part of a claim. A `ClaimContact` object is part of a claim.
- A messaging destination can specify the `Contact` entity as an alternative primary object, in which case that setting applies just to that messaging destination.
- No other entity types can be alternative primary entities for a messaging destination.

Some objects are not associated with any primary object. For example, `Catastrophe` and `User` objects are not associated with a single claim. Messages associated with such objects are called non-safe-ordered messages.

Do not confuse the root object for a message with the primary object associated with a message. The root object is the object that triggered the event. The primary object is the highest-level object related to the root object.

To configure how a message is associated with a primary entity, there are some automatic behaviors when you set the message root object. You can manually set the message root object and the primary entity properties for a message.

Acknowledgement

An acknowledgment is a formal response from an external system to ClaimCenter that declares whether the system successfully processed the message.

- A positive ACK acknowledgment means the external system processed the message successfully.
- A negative NAK acknowledgment means the external system failed to handle the message for some reason.

ClaimCenter distinguishes between the following types of errors.

- An error that throws an exception during the initial sending of the message. Such an error typically indicates a network problem or other retryable issue. ClaimCenter will try multiple times to resend the message.
- A NAK error reported from the external system. ClaimCenter does not resend a message that receives a NAK error.

Safe ordering

Safe ordering is a messaging feature with the following characteristics.

- For each messaging destination, messages are grouped based on their associated primary object.
- In each group, the message's sending order is determined by its time of creation—from the oldest message to the most recent.
- A single message from a group is sent and acknowledged before sending the group's next ordered message.

For example, the default primary entity in ClaimCenter is the `Claim`. Accordingly, messages are grouped based on their associated `Claim` object. However, if a messaging destination sets `Contact` as its alternative primary entity, then the messages for the destination are grouped based on their associated `Contact` object. A destination's alternative primary entity is set in the Messaging editor.

Messages associated with objects other than the primary object are processed as non-safe-ordered messages. In the Messaging editor, non-safe-ordered messages are referred to as **Messages Without Primary**. The logic of how and when to send non-safe-ordered messages differs from that of safe-ordered messages. The behavior can also vary based on the destination's **Strict Mode** setting.

Transport neutrality

ClaimCenter does not assume any specific type of transport or message formatting.

Destinations receive the message any way they want, including but not limited to the following methods.

Submit the message by using remote API calls

Use a SOAP web service interface or a Java-specific interface to send a message to an external system.

Submit the message to a guaranteed-delivery messaging system

For example, submit the message to a message queue.

Save to special files in the file system

For large-scale batch handling, you might send a message by implementing writing data to local text files that are read by nightly batch processes. If you use this technique, you must make your plugins thread-safe when writing to the files.

Send emails

The destination might send emails, which might not guarantee delivery or order, depending on the type of mail system. This approach is acceptable for simple administrative notifications but is inappropriate for systems that rely on guaranteed delivery and message order, which includes most real-world installations.

Overview of messaging flow

The operations of event and message generation and processing are described in the following steps.

The various operations are applied to a hypothetical situation where you want to detect new checks so you can send the information to a check printing system.

1. Application startup – At application startup, ClaimCenter checks its configuration information and constructs messaging destinations. Each destination registers for specific events for which it wants notifications.
In the hypothetical situation, the destination registers for the `CheckAdded` and `CheckChanged` events.
2. Users and APIs trigger events – Events trigger after data changes. For example, a change to data in the user interface, or an outside system calls a web service that changes data. The event represents the changes to application data as the entity commits to the database.
3. Event Fired rule set is executed – ClaimCenter runs the appropriate Event Fired rule set for each message destination defined for the event. triggers. A rule set can choose to generate new messages. Messages have a text-based message payload.

For example, you might write rules that check if the event name is `CheckAdded`. When such an event is identified, the rules might generate a message with an XML payload that describes the added check.

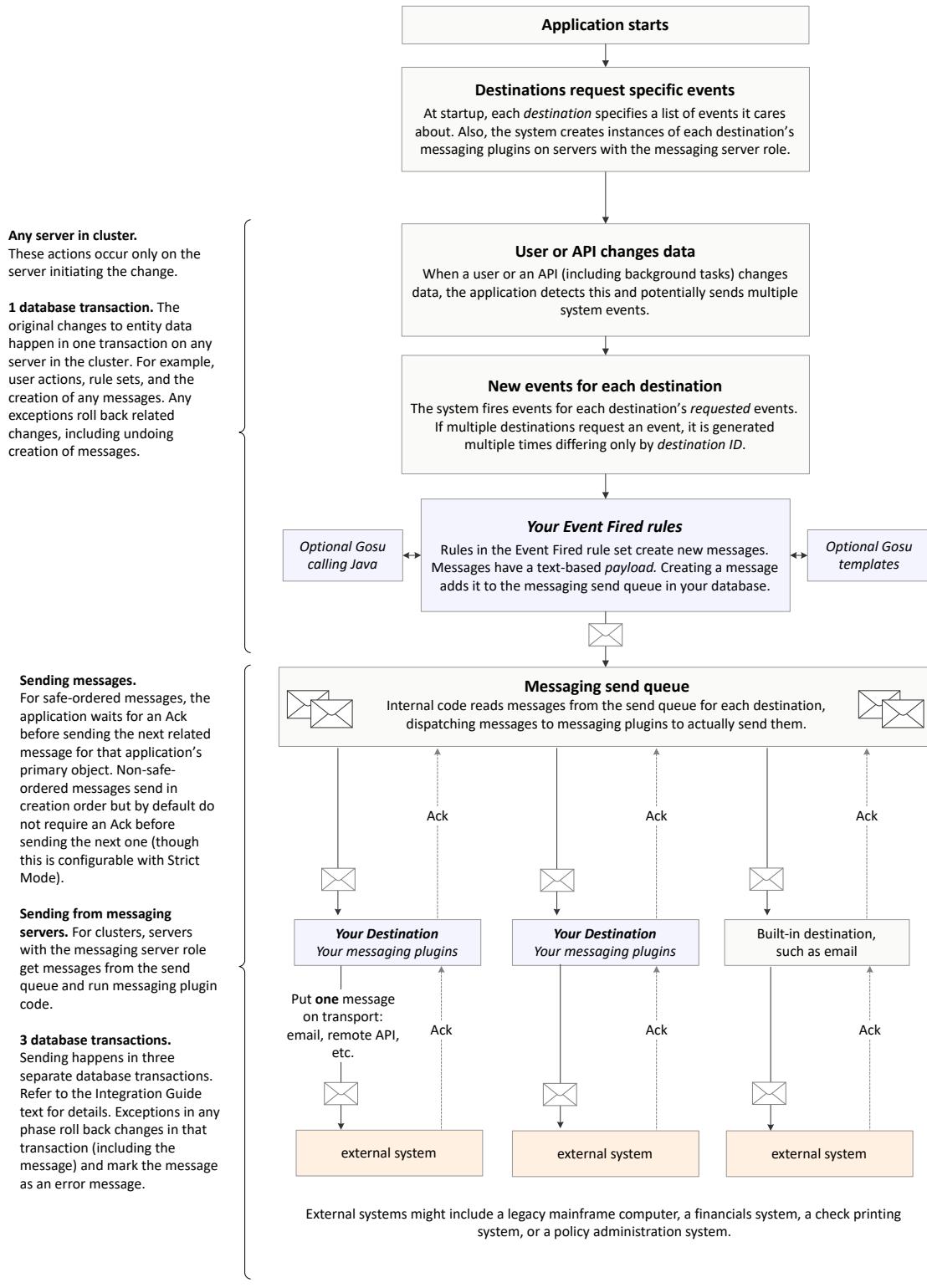
4. ClaimCenter sends message to a destination – Messages are put in a queue and handed one-by-one to the messaging destination.
In ClaimCenter, a check printing destination might take an XML payload and submit the message to an external message queue to notify the external system of the check.
5. ClaimCenter waits for an acknowledgment – The external system replies with an acknowledgment to the destination after it processes the message, and the destination's messaging plugins process this information. If the message was successfully sent, the messaging plugins submit an ACK, and ClaimCenter sends the next message.

Messaging integration code is contained in Event Fired rule sets and `MessageTransport` plugin implementations. After a messaging plugin implementation class is written, it is registered in Studio. Register the messaging plugin implementation first in the **Plugins** editor and then in the **Messaging** editor. For plugin interfaces that can have multiple implementations, such as all messaging plugin interfaces, Studio asks you to name the plugin implementation when registering the plugin class. Use the plugin implementation's class name when configuring the messaging destination in the **Messaging** editor.

Messaging flow details

The following diagram illustrates the chronological flow of events and messaging.

Messaging Overview



Following is a detailed chronological flow of event-related actions.

1. Destination initialization at system startup.

After the ClaimCenter application server starts, the application initializes all destinations. At system startup, ClaimCenter saves a list of events that each destination requested notifications for. If you change the list of events or any destinations after startup, you must restart ClaimCenter so the list can be updated.

Each destination encapsulates all the necessary behavior for that external system, but uses three different plugin interfaces to implement the destination. Each plugin handles different parts of what a destination does.

- The message request plugin handles message pre-processing.
- The message transport plugin handles message transport.
- The message reply plugin handles message replies.

Register new messaging plugins in Guidewire Studio first in the Plugins editor. When you create a new implementation, Studio prompts you for a plugin interface name and, in some cases, for a plugin name. Specify the same plugin name in the Messaging editor in Studio when registering each destination.

Note: Each plugin must be registered in both the Plugins editor and the Messaging editor.

2. A user or an API changes something.

A user action, API call, or a batch process changes data in ClaimCenter.

For example, ClaimCenter triggers an event if you add an exposure to a claim, add a note to anything, or issue a payment.

Note: The change does not fully commit to the database until all new messages are successfully sent to the send queue. Any exceptions that occur before that successful operation will roll back all operations performed for the database transaction.

3. ClaimCenter generates messaging events.

A single action might trigger more than one event. ClaimCenter checks whether each destination has listed each relevant event in its messaging configuration. For each messaging destination that listens for that event, ClaimCenter calls your Event Fired business rules. If multiple destinations want notifications for a specific event, ClaimCenter duplicates the event for each destination that wants that event. To the business rules, these duplicates look identical, except with different destination ID properties.

Note: A change to ClaimCenter data might generate events for one destination, but not for another destination. To change the list of destinations, in the Messaging editor in Studio, select the row for your destination. Additionally, only entity types defined with the `<events>` tag generate added, changed, or deleted events.

4. ClaimCenter invokes Event Fired rules.

ClaimCenter calls the Event Fired rule set for each destination-event pair. This action begins the Message Creation phase of the messaging process. The Message Creation phase continues until all messages for the event are successfully created and added to the send queue.

Event Fired rules must check the event name and the messaging destination ID to determine whether to send a message for that event. Your Event Fired rules generate messages by using the Gosu method `MessageContext.createMessage`.

The rule actions choose whether to generate a message, the order of multiple messages, and the text-based message payload. Rules can use the following techniques:

- Optionally, export an entity to XML by using generated XSDs.

Studio includes the Guidewire XML (GX) modeler tool that helps you export business data entities and other types, like Gosu classes, to XML. Custom XML models can be created that contain only the subset of entity object data that is appropriate for a particular integration point.

You can export an XSD to describe the defined data subset. Then, you can edit your Event Fired rules to generate a payload for the custom entity that conforms to your custom XSD. GX models can also be used to import and parse XML data into in-memory objects.

- Optionally, use Gosu templates to generate the payload.

Rules can use templates with embedded Gosu to generate message content.

- Optionally, use Java classes called from Gosu to generate the payload.

Rules can use Java classes to generate the message content from Gosu business rules.

- Perform optional late binding.

You can use a technique called *late binding* to include parameters in a message payload at message creation time, but evaluate them immediately before sending.

5. New messages are added to the send queue.

After all rules run, ClaimCenter adds any new messages to the send queue in the database. Atomically, the submission of messages to the send queue is part of the same database transaction that triggered the event.

- If all related messages successfully enter the send queue, the transaction succeeds.

- If any operation in the transaction fails, the entire transaction rolls back, including all messages added during the transaction.

The Message Creation phase ends after all messages for the event have been added to the send queue. At this point, the Message Send phase begins.

The length of time that a message might wait in the send queue is dependent on the state of acknowledgments and the status of safe ordering of other message.

6. On servers with the messaging server role, ClaimCenter dispatches messages to messaging destinations.

ClaimCenter retrieves messages from the send queue and dispatches messages to messaging plugins for each destination.

To send a message, ClaimCenter finds the messaging destination's transport plugin and calls its `send` method. The message transport plugin sends the message in whatever native transport layer is appropriate.

If the `send` method throws an exception, ClaimCenter automatically retries the message.

7. Acknowledging messages.

Some destination implementations detect success or failure immediately during sending. For example, a messaging transport plugin might call a synchronous remote procedure call on a destination system before returning from its `send` method.

In contrast, a messaging destination might need to wait for an asynchronous, time-delayed reply from the destination to confirm that it processed the message successfully. For example, it might need to wait for an incoming message on an external messaging queue that confirms the system processed the message.

In either case, the messaging destination code or the external system must confirm that the message arrived safely by submitting an acknowledgment (ACK) to ClaimCenter. Alternatively, it can submit an error, also called a negative acknowledgment (NAK).

You can submit an ACK or NAK in several places.

- For synchronous sending, submit it in your `MessageTransport` plugin during the `send` method.
- For asynchronous sending, submit it in your `MessageReply` plugin. For asynchronous sending, an external system could optionally use a SOAP API to submit an acknowledgment or error.

If using messaging plugins to submit the ACK, you can also make limited changes to data during the ACK, such as updating properties on entities. For financial objects, acknowledgments of messages have special behavioral side effects, such as changing the status of a check.

8. After an ACK or NAK for a safe-ordered message, ClaimCenter dispatches the next related message.

An ACK for a safe-ordered message affects what messages are now sendable. If there are other messages for that destination in the send queue for the same primary object, ClaimCenter soon sends the next message for that primary object.

The Message Send phase of the messaging process ends after all messages for an event have been dispatched and processed.

Overview of message destinations

To represent each external system that receives a message, you must define a *message destination*. Typically, a destination represents a distinct remote system. However, you could use destinations to represent different remote APIs or different message types to send from ClaimCenter business rules. Define a messaging destination in the Guidewire Studio™ Messaging editor. To view the Messaging editor, navigate to **configuration > config > Messaging**, and then open the `messaging-config.xml` file.

If there are several related remote systems or APIs, choose whether they are logically one messaging destination or multiple destinations. Your choice affects messaging ordering. The ClaimCenter messaging system ensures there is no more than one in-flight message per primary object per destination. Therefore, the definition of a destination is critical for predictable message ordering, multithreading, and distributed actions.

Each destination specifies a list of events for which it requires notifications and various other configuration information. Additionally, a destination encapsulates a list of your plugins that perform its main destination functions. The following table compares the types of plugin interfaces that you can configure for a messaging destination.

Message lifecycle phase	Description
Message creation	<p>In Guidewire Studio, write new Event Fired rules that create one or more new messages. The Event Fired code runs on individual application servers that respond to data changes due to user actions, batch processes, or web services. Each new message is a Message entity instance. The new Message entity instance commits within the same database transaction as the changes to the data that triggered the event.</p>
Before send	<p>If you need to transform the <code>Message.Payload</code> property into another format as required by your messaging transport plugin, you can optionally write and register a plugin to handle this transformation. For example, this code might take simple name/value pairs in the message payload and construct a large, complex XML message in a format required by your messaging transport plugin.</p> <p>There are two ways to implement this task.</p> <ul style="list-style-type: none"> • You can write a <code>MessageRequest</code> plugin implementation for this task, in which case it must handle processing both before and after sending. In the Messaging editor, register it in the Request Plugin field. • If you enable Distributed Request Processing for the destination, you can write a <code>MessageBeforeSend</code> plugin implementation to use instead. In the Messaging editor, register it in the Before Send Plugin field, in which case the destination ignores the Request Plugin field. <p>By default, the task runs on the single server that handles this messaging destination. If Distributed Request Processing is enabled for the destination, the task is distributed across multiple servers in the cluster.</p>
Sending	<p>Sends a message, typically to an external system. The underlying protocol might be an external message queue, web service request to external systems, FTP, or a proprietary legacy protocol. Send your messages in your own implementation of the <code>MessageTransport</code> plugin interface. This plugin interface is the only one associated with a messaging transport that is strictly required. Multiple messaging destinations can use the same messaging transport plugin implementation.</p>
After send	<p>If necessary, you can perform post-processing actions on the <code>Message</code> object immediately after sending.</p> <p>There are two ways to implement this task.</p>

Message lifecycle phase	Description
	<ul style="list-style-type: none"> You can write a <code>MessageRequest</code> plugin implementation for this task, in which case it must handle processing both before and after sending. In the Messaging editor, register it in the Request Plugin field. Optionally, you can write a <code>MessageAfterSend</code> plugin implementation to use instead. In the Messaging editor, register it in the After Send Plugin field, in which case the destination ignores the Request Plugin field. <p>Do not use this plugin for asynchronous message replies. Instead, use a <code>MessageReply</code> plugin to handle replies.</p> <p>This task runs only on the single server that handles this messaging destination. The server must have the messaging server role. If more than one server has the messaging server role, the cluster grants a lease to one server to handle messaging for that one destination.</p>
Asynchronous replies	<p>If a destination requires an asynchronous callback for acknowledgments, implement the <code>MessageReply</code> plugin.</p> <p>This task runs only on the single server that handles this messaging destination. The server must have the messaging server role. If more than one server has the messaging server role, the cluster grants a lease to one server to handle messaging for that one destination.</p> <p>ClaimCenter includes multithreaded inbound integration APIs that you can optionally use in conjunction with <code>MessageReply</code> plugins. For example, listen for JMS messages or process text files that represent message replies. If you want to use input data other than JMS messages or text files for message replies, write a custom integration by implementing the <code>InboundIntegrationMessageReply</code> plugin. <code>InboundIntegrationMessageReply</code> is a subinterface of <code>MessageReply</code>.</p>

After you write code that implements a messaging plugin, you must register it in multiple ways. First, register the plugin implementation in Guidewire Studio in the Plugins editor. Go to **configuration > config > Plugins > registry**, and then right-click and choose **Plugin**. Studio prompts you for a plugin name and a plugin interface. The plugin name is a short name that uniquely identifies this plugin implementation.

You must register every plugin implementation in the Plugins editor before using the Messaging editor to specify a messaging plugins. To specify a plugin implementation in the Messaging editor, specify the short plugin name, not the fully-qualified class name.

An implementation of a messaging plugin can be assigned only to a single messaging destination. Do not assign a particular messaging plugin to multiple destinations. This restriction applies to all messaging plugin types, such as `MessageTransport` and `MessageRequest`. Sharing a messaging plugin between multiple destinations can create issues, especially if the plugin overrides the execution methods of the `MessagePlugin` interface, such as `suspend`, `resume`, and `shutdown`.

Use the Messaging editor to create new messaging destinations

The default server on which to run a destination's messaging operations is defined in the **Default Server** field located at the top of the Messaging editor. The server can be specified by either **Host Name** or **Role**. Only a single host name or server role can be specified. In the ClaimCenter base configuration, the default server role is the **messaging** role.

A list of defined destinations is shown in the Messaging editor's left pane. To add a new destination, click the plus icon. To remove the selected destination from the list, click the minus icon. The configuration fields of the selected destination are shown in the right pane. The destination configuration fields are described below.

- ID** – The destination's unique numeric ID. Valid ID range is 0 through 63, inclusive. Guidewire reserves all other destination IDs for built-in destinations, such as the email transport destination.

The ID is typically used in Event Fired rules to check the intended messaging destination for the event notification. If five different destinations request an event that fires, the Event Fired rule set triggers five times for that event. They differ only in the destination ID property (`destID`) within each message context object.

Each messaging plugin implementation must have a `setDestinationID` method. The method stores the ID in a private variable for subsequent retrieval. Configuration code can use the stored value for logging messages or for sending to external systems so that they can programmatically suspend/resume the destination if necessary.

- **Disable destination** – If this check box is selected, the messaging destination is disabled. A destination is enabled by default.
- **Environment** – Environments in which the destination is active. Multiple comma-separated environments can be specified. An environment name cannot include a space character.
- **Name** – Required. The destination name that is used in the ClaimCenter user interface. This can be a display key expression.
- **Server** – Optional. The server on which to run the destination's messaging operations. The server can be specified by either **Host Name** or **Role**. Only a single host name or server role can be specified. If a server is not specified then the **Default Server** is used.
- **Transport Plugin** – Required. The plugin name as specified in the Plugins editor for a `MessageTransport` plugin implementation. The **Transport Plugin** field is the only required plugin name field in the editor. If the messaging destination is enabled then its Transport Plugin must also be enabled. If the Transport Plugin is disabled, the ClaimCenter server will not start.
- **After Send Plugin** – Optional. The plugin name for an implementation of the `MessageAfterSend` plugin interface.
- **Request Plugin** – Optional. The plugin name for an implementation of the `MessageRequest` plugin interface.
- **Reply Plugin** – Optional. The plugin name for an implementation of the `MessageReply` plugin interface.
- **Alternative Primary Entity** – Optional. An alternative primary entity.
- **Chunk Size** – The maximum number of messages for a query to retrieve from the send queue. Default value is 100,000.
- **Number Sender Threads** – Size of the destination's shared thread pool. To send messages, ClaimCenter can create multiple sender threads to distribute the destination's workload. These threads call the messaging plugins that send the messages. The size of the thread pool has a significant effect on messaging performance. Default value is one.
- **Poll Interval** – The amount of time in milliseconds to elapse between the initiations of consecutive message-processing cycles. A message-processing cycle includes retrieving a group of messages from the send queue and processing and sending the messages. Default value is 10,000 (ten seconds).
- **Shutdown Timeout** – Messaging plugins have methods to handle the administrative commands suspend, resume, and preparing for the messaging system to shutdown. The shutdown timeout value is the length of time to wait before attempting to shutdown the messaging subsystem. Default value is 30,000.
- **Max Retries** – The number of automatic retries (`maxretries`) to attempt before suspending the messaging destination. Default value is three.
- **Initial Retry Interval** – The amount of time in milliseconds (`initialretryinterval`) after a retryable error to retry a sending a message. Default value is 1000.
- **Retry Backoff Multiplier** – The amount to increase the time between retries, specified as a multiplier of the time previously attempted. For example, if the last retry time attempted was five minutes, and the multiplier (`retrybackoffmultiplier`) is set to two, ClaimCenter attempts the next retry in ten minutes. Default value is two.
- **Message Without Primary** – Configure the processing of non-safe-ordered messages.
 - **Multi Thread** – Send non-safe-ordered messages asynchronously in a non-deterministic order in multiple threads.
 - **Single Thread** - Send non-safe-ordered messages asynchronously in a strict order in a single thread. Default setting.
 - **Strict Mode** – Send non-safe-ordered messages synchronously in a strict order in a single thread.
- **Distributed Request Processing** – Optional. Distributes the before-send processing across multiple servers in the cluster. The before-send processing is handled by the plugin specified by name in the **Request Plugin** or **Before Send Plugin** field. If you select this checkbox, you can edit the following related fields.

- **Server** – Optional. The server on which to run the destination’s before-send operations. A destination’s before-send operations can be executed on a different server than its other messaging operations. The ability to specify different servers to run a destination’s before-send operations and its other messaging operations enables server load to be allocated in many different configurations.

The server can be specified by either **Host Name** or **Role**. Only a single host name or server role can be specified. If a server is not specified then the destination’s before-send operations are run on the same server as its other messaging operations.
- **Chunk Size** – The maximum number of messages for a query to retrieve from the send queue. Default value is 200.
- **Request Processing Nodes** – The number of cluster nodes to provide this service. Default value is one.
- **Request Processing Threads** – The number of threads on each worker node to use for message request processing. Default value is one.
- **Persist Transformed Payload** – Permanently sets and persists the `Message.Payload` property with the return value of `beforeSend`. By default, the check box is selected which enables the described behavior. If the check box is unselected, the destination ignores the return value of the `beforeSend` method, in which case you must persist the result on other `Message` properties.
- **Always Call Before Send** – If checked, force the `Before Send` plugin to be called even on retries where the message has already been bound. Forcing a call to the `Before Send` plugin is desired if an update to the message payload is necessary before retrying to send the message. Default value is unchecked or `false`.
- **Before Send Plugin** – Optional. The name of a plugin implementation that handles before-send processing. This class must implement either the `MessageBeforeSend` or `MessageRequest` interface.
- **Events** – Optional. A list of event names that correspond to the events relevant to this destination. For example, the `UserChanged` and `UserAdded` events might be of interest to this external system. Each event triggers the Event Fired rule set for that destination. One user action could trigger multiple events. If one action creates more than one event that the destination listens for, ClaimCenter runs the Event Fired rules for every combination of event name and destination. For testing and debugging only, you can specify all events with "`(\w)*`".

Sharing plugin classes across multiple destinations

It is possible to implement messaging plugin classes that multiple destinations share. For example, a transport plugin might manage the transport layer for multiple destinations that use the same physical protocol. In such cases, be aware of the following situations.

- The class instantiates once for each destination. The instance is not shared across destinations. However, you still must write your plugin code as threadsafe, since you might have multiple sender threads. The number of sender threads only affects safe-ordered messaging, and is a field in the Messaging editor in Studio for each destination.
- Each messaging plugin instance distinguishes itself from other instances by implementing the `setDestinationID` method and saving the destination ID in a private class variable. Use this later for logging, exception handling, or notification e-mails.

Handling acknowledgments

Due to differences in external systems and transports, there are two basic approaches for handling replies. ClaimCenter supports synchronous and asynchronous acknowledgments, although in different ways.

- Synchronous acknowledgment at the transport layer

For some transports and message types, acknowledging that a message was successfully sent can happen synchronously. For example, some systems can accept messages through an HTTP request or web service API call. For such a situation, use the synchronous acknowledgment approach. The synchronous approach requires that your transport plugin `send` method actually send the message and immediately submit the acknowledgment with the `message method reportAck`.

To handle errors in general, including most network errors, throw an exception in the `send` method. This triggers automatic retries of the message sending using the default schedule for that messaging destination.

For other errors or flagging duplicate messages, call the `reportError` or `reportDuplicate` methods.

- Asynchronous acknowledgment

Some transports might finish an initial process such as submitting a message on an external message queue. However in some cases, the transport must wait for a delayed reply before it can determine if the external system successfully processed the message. The transport can wait using polling or through some other type of callback. Finally, submit the acknowledgment as successful or an error. External systems that send status messages back through a message reply queue fit this category. There are several ways to handle asynchronous acknowledgments, as described later.

For asynchronous acknowledgment, the messaging system and code path is much more complex. In this case, the message transport plugin does not acknowledge the message during its main `send` method.

The typical way to handle asynchronous replies is through a separate plugin called the message reply plugin. The message reply plugin uses a callback function that acknowledges the message at a later time. For example, suppose the destination needed to wait for a message on a special incoming messaging queue to confirm receipt of the message. The destination's message reply plugin registers with the queue. After it receives the remote acknowledgment, the destination reports to ClaimCenter that the message successfully sent.

One important step in asynchronous acknowledgment with a message reply plugin is setting up the callback routine's database transaction information appropriately. Your code must retrieve message objects safely and commit any updated objects, such as the ACK itself and additional property updates, to the ClaimCenter database.

To set up the callbacks properly, Guidewire provides several interfaces.

- A message finder

A class that returns a `Message` object from its message ID (the `MessageID` property) or from its sender reference ID and destination ID (`SenderRefID` and `DestinationID`). ClaimCenter provides an instance of this class to a message reply plugin upon initialization in its `initTools` method.

- A plugin callback handler

A class that can execute the message reply callback block in a way that ensures that any changes commit. ClaimCenter provides an instance of this class to a message reply plugin upon initialization in its `initTools` method.

- Message reply callback block interface

The actual code that the callback handler executes is a block of code that you provide called a message reply callback block. This code block is written to a very simple interface with a single `run` method. This code can acknowledge a message and perform post-processing such as property updates or triggering custom events.

An alternative to this approach is for the external system to call a ClaimCenter web service to acknowledge the message. ClaimCenter publishes a web service called `MessagingToolsAPI` that has an `ackMessage` method. Set up an `Acknowledgement` object with the `MessageID` set to the message ID as a `String`. If it was an error, set the `Error` property to `true`. Pass the `Acknowledgement` as an argument to the `MessagingToolsAPI` web service method `ackMessage`.

Rule sets must never call message methods for ACK, error, or skip

From within rule set code, you must never call any message acknowledgment or skipping methods such as the `Message` methods `reportAck`, `reportError`, or `skip`. Use those `Message` methods only within messaging plugins.

This prohibition also applies to Event Fired rules.

Destinations in the base configuration

Your rule sets can send standard emails and optionally attach the email to the claim as a document. The email APIs in the base configuration use the email destination provided in the base configuration. ClaimCenter always creates this email destination, independent of your configuration settings.

Additionally, in the base configuration, ClaimCenter provides a destination to communicate with the Insurance Services Office (ISO), which is an optional type of ClaimCenter integration. Unlike the email destination, ClaimCenter creates the ISO destination only if a special line in the server config.xml file is present.

Similarly, the ClaimCenter base configuration provides a destination to communicate with the Metropolitan Reporting Bureau, which is a police accident and police report inquiry service. ClaimCenter creates the Metropolitan destination only if a special line in the server config.xml file is present.

Message processing

Message processing cycle

A single message processing cycle consists of a number of connected operations. The **Poll Interval** field of the Messaging editor specifies the amount of time that must elapse between the initiations of consecutive cycles. If a cycle's operations complete before the interval time expires, the application sleeps for the remaining time.

After completing this cycle of message retrieval and sending, if time remains in the polling interval, the messaging system sleeps until the time interval expires.

Message processing and threads

Two main types of threads perform the main operations in the message process cycle:

- A destination's message reader thread queries the send queue for messages to send. A reader thread retrieves messages in the order of their Event Fired creation-time. The reader thread and send queue can exist on a server that has the **messaging** server role.
- A destination's message sender threads send messages by passing the messages to the messaging plugins. The application supports multiple sender threads for each messaging destination. Configure the number of sender threads in the Messaging editor **Number Sender Threads** field.

Each messaging destination has a message reader thread that queries the send queue for messages for that destination only. The maximum number of messages retrieved in a single query is specified by the chunk size. You configure the chunk size in the Messaging editor **Chunk Size** field.

How message processing works

A single message processing cycle consists of the following operations.

The messaging system retrieves a group of messages from the send queue.

First, ClaimCenter queries the send queue for messages associated with a primary object for that destination. This type of message is referred to as a safe-ordered message. The query for safe-ordered messages returns a maximum of one message per primary object. The reason for this behavior is that the messaging system sends safe-ordered messages synchronously. Until the messaging system receives an acknowledgment of a sent message, it blocks the sending of subsequent safe-ordered messages for a destination/primary object pair. To enforce this behavior, the query returns a maximum of one message for a particular primary object. Therefore, if 100 messages exist in the send queue for a primary object, the query returns only one of them.

Next, ClaimCenter queries the send queue for messages not associated with the primary object for that destination. This type of message is referred to as a non-safe-ordered message.

The messaging system processes the retrieved messages and sends each message to its destination.

For each destination, the message sender threads iterate through all non-safe-ordered messages for that destination, passing the messages to the messaging plugins. Configure the sending of non-safe-ordered messages in the Messaging editor **Message Without Primary** field.

After sending all non-safe-ordered messages, the message sender threads send the safe-ordered messages for that destination.

Message performance

The length of the polling interval and the number of message sender threads significantly affect messaging performance. Discovering their optimum values varies for each installation. In general, for installations that send many messages for each primary entity for each destination, the polling interval has the most significant effect on performance. In contrast, for installations that send many messages, but few for each primary entity for each destination, the number of message sender threads has the most impact on performance.

Configuration parameter `LockPrimaryEntityDuringMessageHandling`

For messages associated with a primary entity (safe-ordered messages), it is possible to optionally lock the primary entity at the database level during messaging operations. Locking the entity can reduce certain edge-case problems in which other threads attempt to modify objects associated with the primary entity. Configuration parameter `LockPrimaryEntityDuringMessageHandling` in file `config.xml` controls the locking mechanism. If the parameter is set to true, ClaimCenter locks the primary entity while performing various message-processing operations.

Sending non-safe-ordered messages

Some messages are not associated with a primary object. Examples include `Claim` or `Contact`.

A message that is not associated with a primary object is called a non-safe-ordered message or message without a primary. By default, messages for the following events are non-safe-ordered.

- Catastrophe events
- Group events
- User events

Settings exist in the Messaging editor **Message Without Primary** field to configure non-safe-ordered messages.

- **Multi thread** – Send non-safe-ordered messages asynchronously in a non-deterministic order in multiple threads.
- **Single thread** – Send non-safe-ordered messages asynchronously in a strict order in a single thread. Default setting.
- **Strict Mode** – Send non-safe-ordered messages synchronously in a strict order in a single thread.

Regardless of the selected option, all non-safe-ordered messages for a particular destination are sent before sending any safe-ordered message to it.

The **Single thread** and **Multi thread** options exhibit the following behaviors.

- The application does not wait for a message acknowledgment before sending subsequent messages.
- With the **Single thread** option, non-safe-ordered messages are sent in the order of their Event Fired creation time among the other messages in a database commit.
- With the **Multi thread** option, the precise order in which messages are sent is non-deterministic.
- Message-sending errors do not block the sending of subsequent messages. The message destination must be able to handle such situations. For example, if a create-object message fails, a subsequent message that assumes the existence of the object might cause a problem unless the destination can handle the situation.

The **Strict Mode** option exhibits the following behaviors.

- The application waits for a message acknowledgment before sending subsequent messages.
- Non-safe-ordered messages are sent in the order of their Event Fired creation time among the other messages in a database commit.
- Message-sending errors block the sending of subsequent messages until an administrator resolves the problem. For example, an administrator might resync the failed message.

Sending safe-ordered messages

For each primary object and message destination pair, safe-ordered messages are sent synchronously. A message acknowledgment must be received before ClaimCenter sends the next safe-ordered message.

Sending safe-ordered messages synchronously prevents possible errors that might occur if messages that are related to each other are sent out of order. For example, suppose an external system must process a parent object-creation message before receiving messages for related child sub-objects. If the messages related to sub-objects are sent before the parent-creation message, the parent object will be unknown to the message-receiving external system.

Safe ordering has significant implications for messaging performance. Suppose the send queue contains ten messages, where each message pertains to a unique and unrelated claim. ClaimCenter can send these ten unrelated messages immediately, rather than synchronously. However, if the send queue contains ten messages for the same claim, ClaimCenter must synchronously send a single message and wait for acknowledgment before sending the next.

Similarly, ContactManager supports the safe ordering of messages that relate to a particular ABContact entity. Each message is sent synchronously for the ABContact and message destination pair.

Guidewire recommendation

To ensure safe-ordering of messages, Guidewire recommends that you set the value of both `message.Claim` and `message.Contact` directly. This ensures that the messages for a particular entity type are safe-ordered on the root `Claim` or `Contact` of which the entity is part.

Improving messaging performance for Oracle databases

There is a `UseOracleHintsOnMessageQueries` parameter in `config.xml`. The base configuration parameter has a default value of `true`. When set to `true`, Oracle databases usually experience better performance due to Oracle hints on application queries for the next chunk of `Message` objects. If you do not want to use this feature, set the `UseOracleHintsOnMessageQueries` parameter in `config.xml` to `false`.

Messaging database transactions during sending

All steps up to and including adding the message to the send queue occur in one database transaction. This is the same database transaction that triggered the event. In addition, there are special rules about database transactions during message sending at the destination level.

Rules for before-send processing

The following rules apply to before-send processing:

- ClaimCenter runs the before-send processing in one database transaction and commits changes, assuming that no exceptions occurred.
- If you enable Distributed Request Processing for a destination, before-send processing can run on multiple cluster members.
- The before-send processing is the `beforeSend` method in an instance of either `MessageRequest` or `MessageBeforeSend` plugin interfaces.
- In your `beforeSend` method, set `Message.Bound` property to `true` to indicate that you handled before-send processing for this message.
- Your `beforeSend` method must be idempotent, which means that it must always return the same result after multiple calls with the same arguments. As needed, check the value of the `Message.Bound` property.
- If the message is retried, the application calls your `beforeSend` method again, even if the `Message.Bound` property has the value `true`.

Rules for send and after-send processing

The following rules apply to send and after-send processing:

- Send and after-send processing are performed on the single cluster member with a lease for this messaging destination. The candidate servers for obtaining a lease for a messaging destination have the `messaging` server role.
- If you enable Distributed Request Processing for a destination, send and after-send processing potentially happens on a different server than before-send processing.

- At message send time, even if you enable Distributed Request Processing for a destination, it is possible that the before-send processing has not yet run for this message. At message send time, if either the message is being retried or `Message.Bound` is `false`, the application runs the before-send processing before attempting to send the message. The before-send processing runs in a separate database transaction from the send and after-send processing.
- ClaimCenter runs both `send` and `afterSend` methods in a single database transaction and commits the changes, assuming that no exceptions occurred.
- ClaimCenter calls the `MessageTransport` plugin method `send` and then runs the after-send processing.
- The after-send processing is the `afterSend` method in an instance of either `MessageRequest` or `MessageAfterSend` plugin interfaces.

Rules for asynchronous replies

The following rules apply to message reply processing for asynchronous replies:

- The `MessageReply` plugin, which optionally handles asynchronous acknowledgments to messages, does its work in a separate database transaction and commits changes, assuming that no exceptions occurred.
- In the default configuration, this work is performed only on the single cluster member with a lease for this messaging destination. The server that handles this messaging destination can be different at message reply time, which can be long after the message was sent.

About database locking during message transactions

To understand database locking during message processing, it is important to understand the meanings of the following terms:

Term	Description
Primary entity	A primary entity represents a type of high-level object that a Guidewire application uses to group and sort related messages. The default primary entity in Guidewire ClaimCenter is <code>Claim</code> .
Primary object	A primary object is a specific instance of a primary entity, which ClaimCenter stores in table <code>cc_claim</code> .
Root object	A root object is the object that triggered the message event. Do not confuse the root object with the primary object. If an activity triggers a message event, then the root object is an <code>activity</code> object. ClaimCenter stores the activity root object in table <code>cc_activity</code> .
Message object	A message event generates a message object. ClaimCenter stores a message object instance in table <code>cc_activity</code> .

During each message transaction, the messaging system does one of the following:

- The messaging system performs row locking in the database for the primary object associated with the message, unless configured otherwise.
- Or, the messaging system performs row locking in the database on the root and message objects.

The intent of locking database rows is to prevent other transactions, users, or activities from modifying the content of the message during message processing.

Database locking during non-distributed message transactions

During the preprocessing phase of a *non-distributed* message transaction, the messaging system locks the transaction's message object stored at the database level in table `cc_message`. It also locks the direct (root) entity associated with the message, an instance of an activity object (in `cc_activity` table), for example. The messaging system also blocks code that accesses the locked message until the messaging system releases the object.

Database locking during distributed message transactions

In contrast, while processing a *distributed* message transaction, the messaging system does not automatically lock the transaction's message object (in table `cc_message`). However, it is possible configure the message system to lock the message object during distributed transactions by setting configuration parameter `LockDuringDistributedMessageRequestHandling` in file `config.xml` to true.

Database locking of primary entity instances

Unless configured to not do so, the message system locks the primary entity associated with the message by default. For example, suppose that a change to an activity tied to a Claim triggers a message event. The message system then locks the primary object associated with the message, Claim in `cc_claim` table.

You configure the locking of a primary entity instance by setting configuration parameter `LockPrimaryEntityDuringMessageHandling` in `config.xml` to either `true` or `false`. The default value for `LockPrimaryEntityDuringMessageHandling` is `true`, which means the message system automatically locks the primary entity during a message transaction. Setting this parameter to `false` disables this behavior. The lock applies only to the primary entity instance and not to any sub-objects. Again, the messaging system blocks code that accesses a locked entity instance from running until the messaging system releases the object.

However, regardless of how you set `LockPrimaryEntityDuringMessageHandling`, the messaging system locks the primary entity instance only if the transaction's message object is also locked. For example, a distributed message transaction that does not lock its message object does not lock the primary entity either, even if locking of the entity is enabled by the `LockPrimaryEntityDuringMessageHandling` parameter.

Locking and unlocking primary entities

If you enable message or primary entity locking, the messaging system locks the objects during each of the following operations and unlocks the objects at the operation's completion:

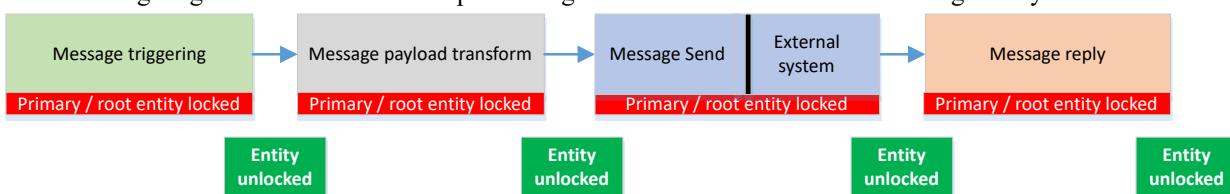
- While the `beforeSend` method executes
- While the `send` method executes
- While processing message-reply operations
- While marking a message as skipped

Locking and concurrent data exceptions

Message and primary entity locking operations exist outside the concurrent data exception system. ClaimCenter disables concurrent data exception checking during the messaging operation.

About table row locking

The following diagram illustrates the multiple message transactions that occur in a message lifecycle



A synchronous message cycle consists of the following defined transactions:

- A transaction that triggers a message event
- A transaction that transforms the event information into the message payload
- A transaction that sends the message payload to an external message processing system
- A transaction in which the external message processing system sends back a reply to the message

An asynchronous message cycle consists of fewer transactions as the message system does not wait for a reply:

- A transaction that triggers a message event
- A transaction that transforms the event information into the message payload
- A transaction that sends the message payload to an external message processing system

Table locking

During each of these defined transactions, the message system automatically locks the following rows in the database:

- The row in database table `cc_message` for the message object
- The row in the database table for the instance of the direct object for the message, `cc_activity` for an activity object, for example.

Configuration parameter `LockPrimaryEntityDuringMessageHandling` affects this behavior:

<code>LockPrimaryEntityDuringMessageHandling</code> Behavior	Description
<code>true</code>	The messaging system locks the primary object of the message. For example, for a message triggered by an activity associated with a claim, the message system locks the appropriate row in table <code>cc_claim</code> .
<code>false</code>	The messaging system does not lock the primary object, but it does lock the table row for the message object itself (<code>cc_message</code>) and the row in the database for the instance of the direct object that the message effects

Thus, for non-distributed message transactions, the message system does one of the following, depending on how you set `LockPrimaryEntityDuringMessageHandling`:

- The message system locks the primary entity row in the database, but, it does not lock table rows for the direct (root) entity and message object.
- The message system locks table rows for the direct (root) entity and message object, but, it does not lock the primary entity associated with the message.

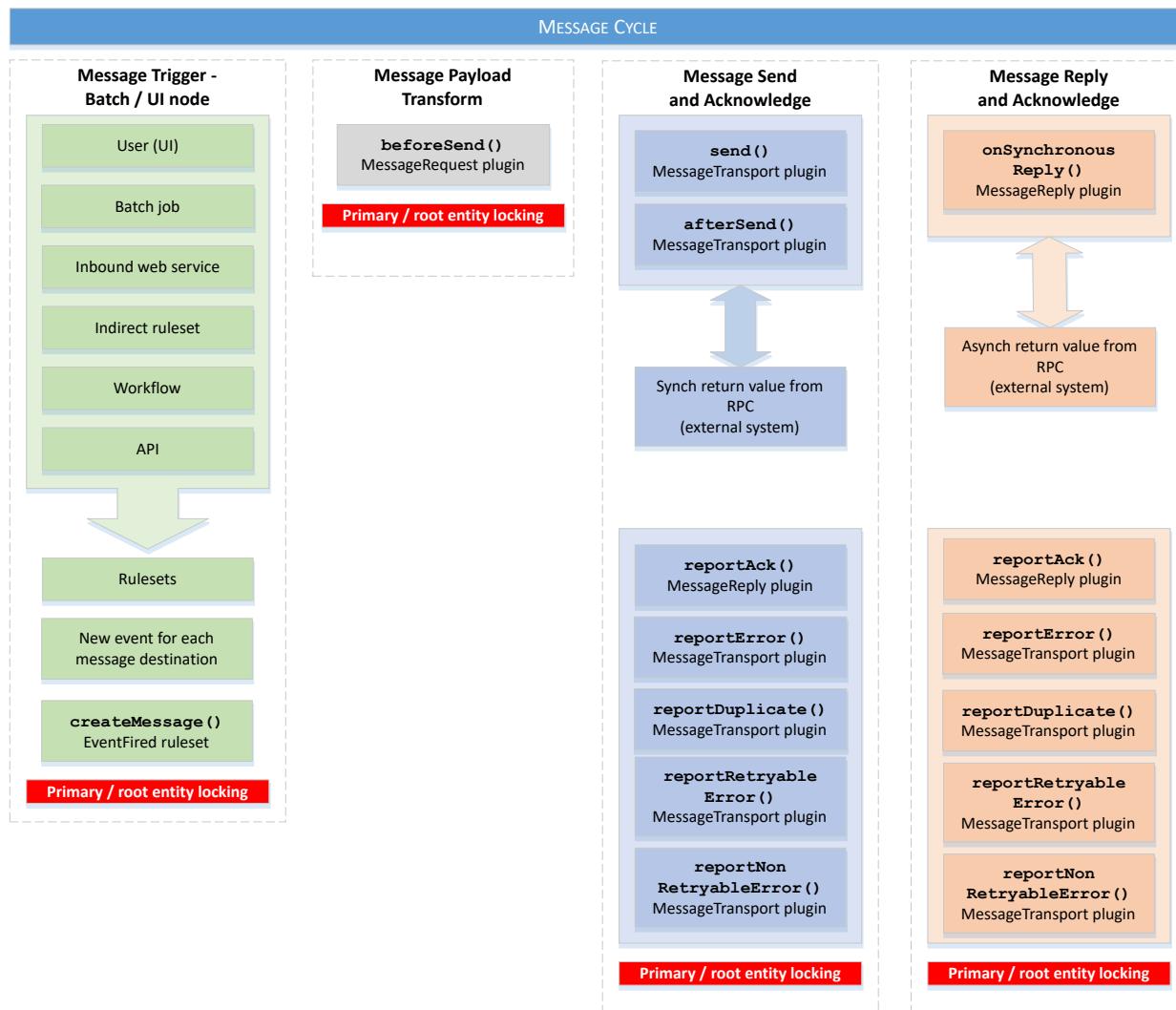
The intent of locking database rows is to prevent other transactions, users, or activities from modifying the content of the message during message processing. Locking the primary object ensures that only one message can affect the claim at a time.

Table unlocking

The message system unlocks the locked table rows at the end of each transaction.

Detailed description of database locking in messaging

The following graphic provides a detailed view of what happens during a message transaction.



Message transactions

The graphic shows four database transactions that occur during the message lifecycle:

1. Message triggering
2. Message payload transformation
3. Message send and acknowledgment
4. Message reply and acknowledgment

During each transaction, the message system (or an external system) performs a number of actions such as executing code or managing errors that occurred during the transmission or handling of the message.

Database locking behavior

Each message transaction blocks any change (through row-locking in the database) to one of the following objects associated with the message:

- Either the ClaimCenter primary object
- Or the root object directly related to the message

Configuration parameter `LockPrimaryEntityDuringMessageHandling` governs this behavior.

During each message transaction, the message system locks the affected object (through database row locking) for the duration of that transaction. At the completion of each transaction, the message system unlocks the table row associated with the affected object.

For synchronous messaging, database row locking happens three times during the message lifecycle. For asynchronous messaging, database row locking happens four times during the message lifecycle.

Database view of message processing

The ClaimCenter `cc_Message` and `cc_MessageHistory` tables contains a number of columns that store timestamp information related to message creation and processing. Identically named columns in the two tables contain identical information. The `MessageHistory` table contains one more message-related column than the `Message` table.

The following table shows the general steps involved in processing a message along with the database tables and columns that stores relevant information for that step. The first two steps can take place on a server with either the ui or batch server role.

Step	Acquiring lock	Lock acquired / performing operation	Unlock and operation completed
1. Message creation		Inherits database lock created in the EventFired rule that created the message payload.	<code>cc_Message.CreationTime</code>
2. Payload transformation	<code>cc_Message.BeforeSendLockTime</code> (MessageRequest plugin)	<code>cc_Message.BeforeSendLockedTime</code>	<code>cc_Message.BeforeSendTime</code> These three timestamp values are present only if there is preprocessing being performed on the message destination. Otherwise, all three of these values are null.
3. Message send	<code>cc_Message.SendLockTime</code> (MessageTransport plugin)	<code>cc_Message.SendLockedTime</code>	<code>cc_Message.SendTime</code>
4. Message after send	Inherits database lock in the send method (MessageTransport plugin)		<code>cc_Message.AfterSendTime</code>
5. Message retry			<code>cc_Message.RetryTime</code> The future time to retry the message.
6. Message reply, acknowledgment	This value is different between synchronous and asynchronous message acknowledgment: <ul style="list-style-type: none">• Synchronous - ClaimCenter acquires the lock. The value is typically that of <code>AfterSendTime</code>.• Asynchronous - ClaimCenter acquires the lock. The value is the current time value.		<code>cc_MessageHistory.ackedMessage</code> The time at which message processing in the <code>MessageReply</code> plugin completes.

In general, use the data in table `cc_MessageHistory` to perform your analysis. The message system populates the data in the `cc_MessageHistory` table after it sends out the message, moving the relevant data from the `cc_Message` table into the `cc_MessageHistory` table at that time.

Table column meanings

The following table describes the table columns listed in the previous table.

Column	Timestamp
CreationTime	The creation time of the message record. At this point, the entity (root or primary) is already locked. Thus, the step does not add additional wait time to the lock/release cycle.
QueryTime	The time at which the MessageTransport plugin queries the database for the record (cc_Message). For the messaging tables: <ul style="list-style-type: none"> • cc_Message - This value is likely to be null, which means that ClaimCenter has yet to pick up the message for processing. • cc_MessageHistory - This value is always present.
BeforeSendLockTime	Time at which the MessageRequest plugin requests a lock on this message record for payload generation.
BeforeSendLockedTime	The time at which the MessageRequest plugin acquires the record lock and is ready to perform payload generation. If there is an extensive amount of time spent at this step, it is due to the locking operation.
BeforeSendTime	The time between payload generation and updating the message record in the database. If there is an extensive amount of time spent in this step, it is likely due to the volume of messages or to coding issues. To calculate the time spent in this step, use the following calculation: $\text{BeforeSendTime} - \text{BeforeSendLockedTime}$
SendLockTime	The time at which the MessageTransport plugin requests a lock of the message record for the Message.send method call.
SendLockedTime	The time at which the MessageTransport plugin acquires the record lock and is ready to perform the Message.send method call. If there is an extensive amount of time spent at this step it is due to the locking operation.
SendTime	The time at which the Message.send method call completes. If there is an extensive amount of time spent at this step, it is due to coding issues or waiting on external integration.
AfterSendTime	The time at which the Message.afterSend method completes execution of customer code and the message exits the MessageTransport plugin.
AckedTime	The time at which the Message.ackMessage method call completes and the reply message exits the MessageReply plugin.
RetryTime	The time scheduled (in the future) to retry sending the message record.

Understanding message history data

The timestamp values that ClaimCenter stores in the Message and MessageHistory tables can provide insights into timing issues with messages and message queues. However, in working with the values in the MessageHistory table, it is important to realize that:

- A message destination uses the same set of message fields for every message sent to that destination.
- A message destination can use a set of message fields that differs from other message destinations.

Message preprocessing

If you decide to implement message preprocessing, you can perform the preprocessing work:

- In the message transport
- In a separate preprocessor message thread

It is also possible to override message preprocessing if you are performing processing in the transport thread as well.

Use the following message timestamp values to help in understanding timing issues with preprocessing messages.

Column	Timestamp
BeforeSendLockTime	The time immediately before the MessageRequest plugin requests a lock on this message record for payload generation.
BeforeSendLockedTime	The time immediately after the MessageRequest plugin acquires the record lock and is ready to perform payload generation.
BeforeSendTime	The time immediately after ClaimCenter executes the <code>Message.beforeSend</code> method, which is the time between payload generation and update of the message record in the database.

Determine lock wait time

To determine how long ClaimCenter has to wait to acquire the message lock, use the following calculation:

$$\text{BeforeSendLockedTime} - \text{BeforeSendLockTime}$$

Determine message preprocessing time

To determine how long the customer code took to execute message preprocessing, use the following calculation:

$$\text{BeforeSendTime} - \text{BeforeSendLockedTime}$$

These statistics can be empty for a message destination if that particular destination does not preprocess messages.

Message send

Use the following message timestamp values to help in understanding the timing issues involved in sending a message.

Column	Timestamp
SendLockTime	The time immediately before the ClaimCenter acquires the message lock and calls the <code>Message.send</code> method.
SendLockedTime	The time immediately after the MessageTransport plugin acquires the record lock and when it is ready to perform the <code>Message.send</code> method call.
SendTime	The time after the <code>Message.send</code> method call completes.
AfterSendTime	The time after the <code>Message.afterSend</code> method completes executing customer code and the message exits the MessageTransport plugin.
AckedTime	The time after the <code>Message.ackMessage</code> method call completes and the reply message exits the MessageReply plugin. This time value is an outlier. ClaimCenter sets this value as it commits the Acked message to the database. In setting this value: <ul style="list-style-type: none"> • If ClaimCenter sets this value asynchronously, the time value does not capture the lock wait time. • If ClaimCenter sets this value after the <code>AfterSendTime</code> value, the time value is typically not meaningful except under extreme load.

Determine lock wait time

To determine how long ClaimCenter waited for the lock (which is likely to be small if preprocessing is also occurring in the transport), use the following calculation:

$$\text{SendLockedTime} - \text{SendLockTime}$$

Determine send code execution time

To determine how long the customer send code took to execute (which includes any request/response processing and waiting on remote responses), use the following calculation:

$$\text{SendTime} - \text{SendLockedTime}$$

Determine afterSend code execution time

To determine the amount of time the customer `afterSend` code took to execute after the `Message.afterSend` method call, use the following calculation:

$$\text{AfterSendTime} - \text{SendTime}$$

Message queues

Use the following timestamp values to help in understanding timing issues involving message queues.

Column	Timestamp
CreationTime	The time at which ClaimCenter creates the message record in the database.
QueryTime	The time when the MessageTransport plugin queries the database for the record, which is the time when ClaimCenter picks up the message as part of the group of message to execute.

Determine queue backlog

To determine how backed up a queue is, use the following calculation:

`QueryTime - CreationTime`

Determine work thread capacity

To determine if there are insufficient worker threads for the amount of work items in the message queue, use one of the following calculations:

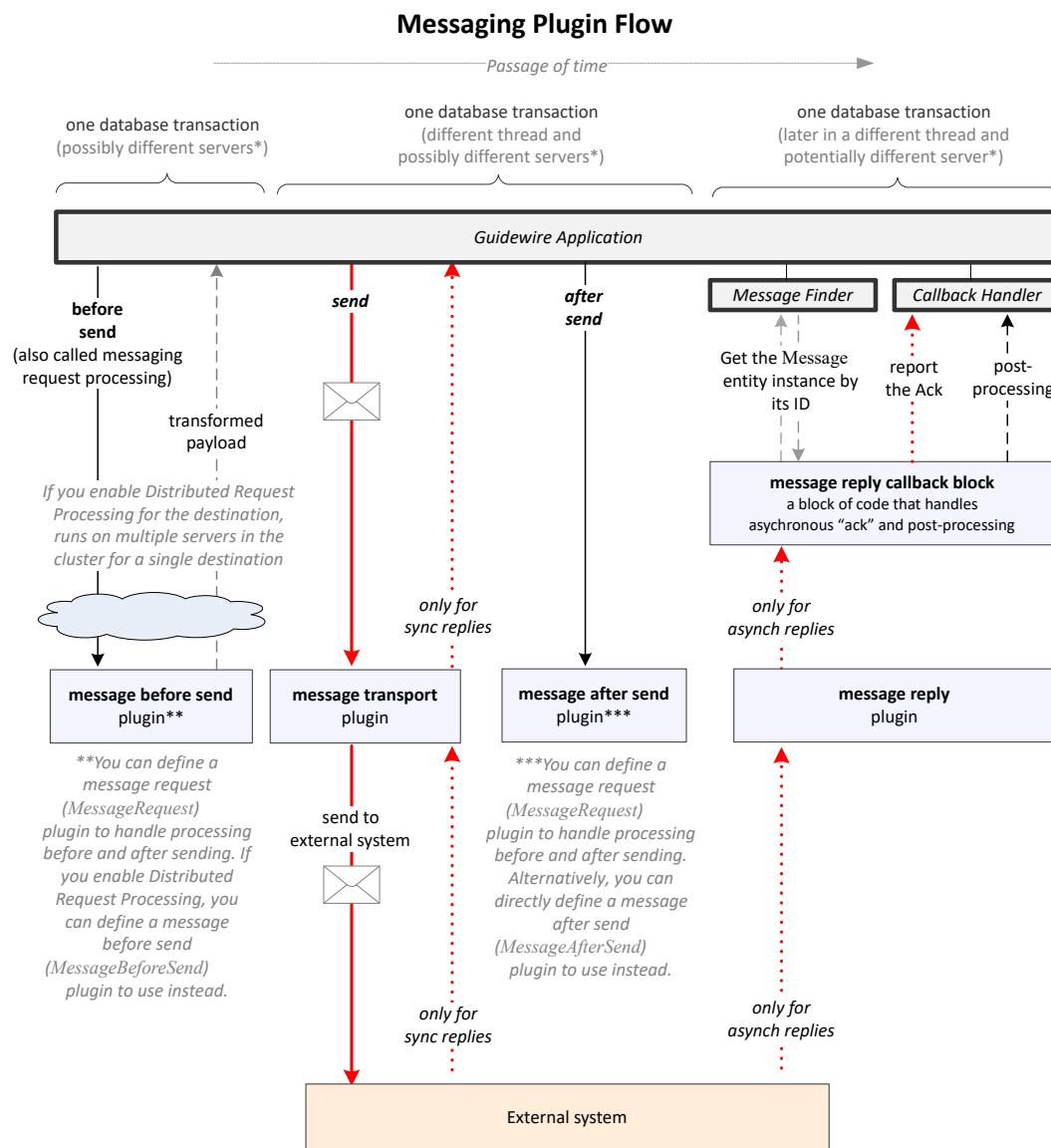
`BeforeSendLocktime - QueryTime`

`SendLockTime - QueryTime`

Use the second calculation if there is no message preprocessing.

Messaging plugin interaction and flow diagram

The following diagram illustrates messaging plugins, and the chronological flow of actions between elements.

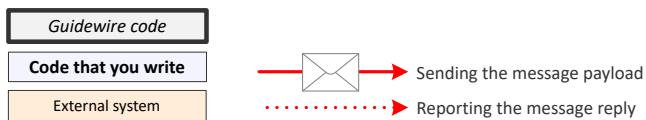


*By default, **before-send** actions run on the one server in the cluster with the lease to handle that destination. If you enable Distributed Request Processing for the destination, these actions run on multiple servers.

The **send** and **after-send** phases run in a different thread on the server that has the lease for that destination. If you enable Distributed Request Processing for the destination, these actions potentially run on an entirely different server from the **before-send** actions.

The **reply** phase happens on the server with the lease for that destination. This happens in a separate thread. This reply might come significantly later, and potentially on a different server if the cluster lease assignments have changed.

KEY



Messaging events in ClaimCenter

ClaimCenter generates events associated with a specific entity instance as the root object for the event.

Sometimes an event root object is a higher-level object such as claim. In other cases, the event is on a subobject, and your Event Fired rules must do some work to determine what high-level object it is about. For example, the Address subobject is common and changing it is common. However, what larger object contains this address? You might need this additional context to do something useful with the event.

For example, was the address a claim's loss location? Was it the claimant's temporary address?

In ClaimCenter, all the child objects of a claim have a `Claim` property that references the claim.

ClaimCenter triggers events for many objects if an entity is added, removed (or retired), or if a property changes on the object. For example, selecting a different claim type on any claim generates a changed event on the claim.

The changes are about the change to that database row itself, not on subobjects. For example, ClaimCenter reports a change to an object if a foreign key reference to a subobject changes but not if properties on the subobject changes.

There are exceptions to this rule.

For example, selecting a different claimant on a claim is a change to the claim. A change to an exposure is an exposure change, but not a claim change.

List of messaging events

The following table describes the events that ClaimCenter raises. In this table, standard events refer to the added, changed, and removed events for entities that generate events. For example, the `Claim` entity would generate events whenever code adds, changes, or removes entities of that type in the database. In those cases, the Event Fired business rules would see `ClaimAdded`, `ClaimChanged`, and `ClaimRemoved` events if one or more destinations registered for those event names.

Entity	Events	Description
Activity	<code>ActivityAdded</code> <code>ActivityChanged</code> <code>ActivityRemoved</code>	Standard events for the <code>Activity</code> entity. <code>ActivityChanged</code> indicates that an activity changed, including marking the activity completed or skipped.
Assignable Entities: • <code>Claim</code> • <code>Exposure</code> • <code>Activity</code> • <code>Matter</code>	<code>AssignmentAdded</code> <code>AssignmentChanged</code> <code>AssignmentRemoved</code>	Assignment events for assignable entities. The assignable entity is the root entity for the event. If an assignment added to this entity, <code>AssignmentAdded</code> triggers. If the entity previously had an assignment and now has no assignment, <code>AssignmentRemoved</code> triggers. If assignment data such as assigned user, group, date changed, <code>AssignmentChanged</code> triggers.
Claim	<code>ClaimAdded</code> <code>ClaimChanged</code> <code>ClaimRemoved</code>	Standard events for claims. Note the following situations. <ul style="list-style-type: none"> Adding an <code>Exposure</code> to a claim does not trigger the <code>ClaimChanged</code> event, even though it might appear that way. Instead, adding a new <code>Incident</code> triggers the <code>ClaimChanged</code> event. The <code>ClaimAdded</code> and <code>ClaimChanged</code> events trigger for claims and exposures independently from their validity level. If you want to send claims and exposures to external systems only after reaching a specific validation level, your Event Fired rules can ignore that event. Your rules can optionally check the claim state, for example detect the state change from open to draft.

Entity	Events	Description
		<ul style="list-style-type: none"> To prevent duplicate messages of certain types (such as initial message to external systems), create data model extensions on claim or exposure and set them within Event Fired rules. In your rules, this change marks a claim or exposure as already sent to that external system. The <code>ClaimChanged</code> event triggers if a claim closes or reopens. ClaimCenter does not trigger these events for a claim resync. See <code>ClaimResync</code> in this table.
	<code>ClaimResync</code>	<p>Resync a claim. This is an administrator request to drop all pending and <i>in error</i> messages for a claim. Then, your Event Fired rules tries to resync the claim with the external system to recover from integration problems. Administrators can request a claim resync from the application user interface or through the web services API.</p>
	<code>PersonalDataPurge</code>	<p>A Claim purge was committed as part of a personal data purge. You must define your own rule to handle this event.</p>
<code>ClaimInfo</code>	<code>ClaimInfoAdded</code> <code>ClaimInfoChanged</code> <code>ClaimInfoRemoved</code>	Standard events for <code>ClaimInfo</code> objects.
<code>Coverage</code>	<code>CoverageAdded</code> <code>CoverageChanged</code> <code>CoverageRemoved</code>	<p>Coverage entities are abstract entities that exist in the form of coverage subtypes, such as <code>PolicyCoverage</code>. <code>CoverageAdded</code>, <code>CoverageChanged</code>, and <code>CoverageRemoved</code> events are sent for coverage subtypes. Most customers do not let users remove existing coverages. Thus, in practice, ClaimCenter may never trigger the <code>CoverageRemoved</code> event. Changing a property on a <code>Coverage</code> entity triggers the <code>Policy_Changed</code> event in addition to the <code>Coverage_Changed</code> event.</p>
<code>Document</code>	<code>DocumentAdded</code> <code>DocumentChanged</code> <code>DocumentRemoved</code>	Standard events for documents. The <code>DocumentAdded</code> events trigger if a document links to the claim file. Most implementations do not permit users to remove documents once added, so this event may never trigger.
	<code>DocumentStore</code>	When the asynchronous document content storage code attempts to store in the database, ClaimCenter triggers a <code>DocumentStore</code> event on the relevant <code>Document</code> entity instance.
	<code>FailedDocumentStore</code>	When the asynchronous document content storage code fails to store in the database, ClaimCenter triggers a <code>FailedDocumentStore</code> event on the relevant <code>Document</code> entity instance. The event is important because it is the only notification

Entity	Events	Description
		that document storage failed. Use this event to create a notification for administrators or users.
Exposure	ExposureAdded ExposureChanged ExposureRemoved	<p>Standard events for exposures.</p> <ul style="list-style-type: none"> If it is important to send claims and exposures to external systems only after reaching a specific validation level, Event Fired rules decide whether to ignore or log that event. Your rules can optionally check the claim state, for example detect the state change from open to draft. To prevent duplicate messages of certain types such as initial message to external systems, create data model extension messaging-specific properties on claim or exposure. Set your properties in Event Fired rules. Your rules indicate that a claim or exposure was already sent to that external system. The ExposureChanged event triggers if a claim is closes or reopens. If your implementation does not allow users to remove exposures once added, the ExposureRemoved event may never trigger.
Matter	MatterAdded MatterChanged MatterRemoved	Standard events for the Matter entity. The MatterAdded event triggers after a legal matter attaches to a claim. If your implementation does not allow users to remove legal matters from a claim once added, the MatterRemoved event may never trigger.
MetroReport	MetroReportAdded MetroReportChanged MetroReportRemoved	Standard events for MetroReport entities.
Negotiation	NegotiationAdded NegotiationChanged NegotiationRemoved	Standard events for Negotiation entities.
Note	NoteAdded NoteChanged NoteRemoved	Standard events for Note entities.
Policy	PolicyAdded PolicyChanged PolicyRemoved	<p>Standard events for policies.</p> <ul style="list-style-type: none"> The PolicyAdded event triggers if something adds a policy snapshot to ClaimCenter. This happens after entering a new claim or after changing the policy on an existing claim. The PolicyChanged event triggers after a policy or any subobject updates. This includes property, vehicle, stat code, but not coverages.

Entity	Events	Description
		<ul style="list-style-type: none"> ClaimCenter sends these events independent of the claim validation level of the associated claim. Filter events as needed. Changes to a property on a Coverage entity triggers the PolicyChanged event in addition to the CoverageChanged event.
PolicyCoverage	CoverageAdded CoverageChanged CoverageRemoved	Standard events for Coverage entities.
PropertyCoverage	CoverageAdded CoverageChanged CoverageRemoved	Standard events for Coverage entities.
VehicleCoverage	CoverageAdded CoverageChanged CoverageRemoved	Standard events for Coverage entities.
ServiceRequest	ServiceRequestAdded ServiceRequestChanged ServiceRequestRemoved	Standard events for ServiceRequest entities.
ServiceRequestDocumentInfo	ServiceRequestDocumentInfoAdded ServiceRequestDocumentInfoChanged ServiceRequestDocumentInfoRemoved	Standard events for ServiceRequestDocumentInfo entities.
ServiceRequestMessage	ServiceRequestMessageAdded ServiceRequestMessageChanged ServiceRequestMessageRemoved ServiceRequestMessageToVendorAdded	Standard events for ServiceRequestMessage entities.
General-purpose events		
ProcessHistory	ProcessHistoryAdded ProcessHistoryChanged ProcessHistoryRemoved	Some integrations can be done as a batch process. For example, the financials escalation process starts manually or on a timer. You could use the event/messaging system to write records to a batch file (or rows to a database table) as each transaction processes. Perhaps during the night, you can submit the batch data to some downstream system. To coordinate this activity, you can use the ProcessHistory entity. As a batch process starts, a ProcessHistoryAdded event triggers. Listen for the ProcessHistoryChanged event in your Event Fired rules, and check the <code>processHistory.CompletionTime</code> property for the datestamp in a <code>datetime</code> object.
SoapCallHistory	SoapCallHistoryAdded SoapCallHistoryChanged SoapCallHistoryRemoved	Standard events for SoapCallHistory entities. The application creates one for each incoming web service call.

Entity	Events	Description
StartablePluginHistory	StartablePluginHistoryAdded StartablePluginHistoryChanged StartablePluginHistoryRemoved	Standard events for StartablePluginHistory entities. The application creates these to track when a startable plugin runs.
InboundHistory	InboundHistoryAdded InboundHistoryChanged InboundHistoryRemoved	Standard events for root entity Invoice.
Administration events		
Catastrophe	CatastropheAdded CatastropheChanged CatastropheRemoved	Standard events for catastrophes. The CatastropheChanged event triggers if the catastrophe retires or gets a new catastrophe code.
Group	GroupAdded GroupChanged GroupRemoved	Standard events for groups. The GroupChanged event triggers after additions or removals of users from a group.
Role	RoleAdded RoleChanged RoleRemoved	Standard events for Role entities.
User	UserAdded UserChanged UserRemoved	Standard events for users. ClaimCenter triggers the UserChanged event only for changes made directly to the user record, not for changes to roles or group memberships. Be aware that changes to the user's contact record such as a phone number cause a ContactChanged event, not a UserChanged event.
UserSettings	UserSettingsAdded UserSettingsChanged UserSettingsRemoved	Standard events for user settings.
GroupUser	GroupUserAdded GroupUserChanged GroupUserRemoved	Standard events for root entity GroupUser.
UserContact	ContactAdded ContactChanged ContactRemoved	Standard events for Contact entities.
Financial transaction events		
BulkInvoice	BulkInvoiceAdded BulkInvoiceChanged BulkInvoiceRemoved	Standard events for the BulkInvoice entity.
	BulkInvoiceStatusChanged	The bulkinvoice.status property changed.
Check	CheckAdded CheckChanged CheckRemoved	Standard events for the Check entity.
	CheckStatusChanged	The check.status property changed.

Entity	Events	Description
Reserve	ReserveAdded	Standard events for reserves. Check for changes to the status property within rules that catch the ReserveChanged event.
	ReserveChanged ReserveRemoved	
AuthorityLimit-Profile	ReserveStatusChanged	The reserve.status property changed.
	AuthorityLimitProfileAdded	Standard events for the AuthorityLimitProfile entity.
	AuthorityLimitProfileChanged AuthorityLimitProfileRemoved	
Payment	PaymentAdded	Standard events for Payment entities. Check for changes to the status property within rules that catch the PaymentsChanged event.
	PaymentChanged PaymentRemoved	
	PaymentStatusChanged	The payment.status property changed, possibly but not necessarily because of a change to the associated check.
RecoveryReserve	RecoveryReserveAdded	Standard events for recovery reserves. The RecoveryAdded event triggers after some code adds a recovery reserve change. Typically the recovery is initially in the submitting status.
	RecoveryReserveChanged RecoveryReserveRemoved	
Recovery	RecoveryReserveStatusChanged	The recoveryreserve.status property changed.
	RecoveryAdded	Standard events for recovery transactions.
	RecoveryChanged RecoveryRemoved	
Transaction	RecoveryStatusChanged	The check.status property changed.
	TransactionAdded	Standard events for Transaction, which is the supertype of other transactions.
	TransactionChanged TransactionRemoved	
RITransaction	RITransactionAdded	Standard events for RITransaction entities.
	RITransactionChanged	
	RITransactionRemoved	
Contacts and address book events		
ABContact	ABContactAdded	Standard events for ABContact entities (ContactManager only).
	ABContactChanged ABContactRemoved	
Adjudicator	PersonalDataPurge	An ABContact purge was committed as part of a personal data purge (ContactManager only). You must define your own rule to handle this event.
	ContactAdded	Standard events for the Contact entity.
	ContactChanged ContactRemoved	
Attorney	ContactAdded	Standard events for the Contact entity.
	ContactChanged	

Entity	Events	Description
	ContactRemoved	
AutoTowingAgcy	ContactAdded ContactChanged ContactRemoved	Standard events for the Contact entity.
AutoRepairShop	ContactAdded ContactChanged ContactRemoved	Standard events for the Contact entity.
ClaimAssociation	ClaimAssociationAdded ClaimAssociationChanged ClaimAssociationRemoved	Standard events for the ClaimAssociation entity.
ClaimContact	ClaimContactAdded ClaimContactChanged ClaimContactRemoved	Standard events for ClaimContact entities. The ClaimContactChanged event is particularly important if you track changes to claim contacts within ClaimCenter. For example, if the Tax ID or phone number of a claimant changed, you might want to update the associated claimant or exposure records in an external system. However, most implementations do not listen for the ClaimContactRemoved event. Instead, handle removals within Event Fired rules for the event ClaimContactRoleRemoved. Contrast this entity with Contact and ClaimContactRole. Also refer to the event for ClaimContactContactChanged later in this table.
ClaimContact	ClaimContactContactChanged	ClaimCenter triggers this if either the contactID changes on the claimContact or if the referenced contact changes.
ClaimContactRole	ClaimContactRoleAdded ClaimContactRoleChanged ClaimContactRoleRemoved	Standard events for ClaimContactRole entities. The ClaimContactRoleAdded event triggers after a contact associates with a new role for a claim, policy, exposure, matter, negotiation, or evaluation. For example, this event triggers after some code adds a new exposure and the insured is now the claimant role for that new exposure. This is an important event if you track contact changes. For many implementations, roles are more important to track than contacts. Similarly, the ClaimContactRoleRemoved event triggers if a contact no longer has a certain role for the claim, policy, exposure, matter, negotiation, or evaluation. For example, if the contact is the main contact for the claim, but the main contact changes to a different person.
CompanyVendor	ContactAdded ContactChanged ContactRemoved	Standard events for Contact entities.
Company	ContactAdded ContactChanged	Standard events for Contact entities.

Entity	Events	Description
	ContactRemoved	
Contact	ContactAdded ContactChanged ContactRemoved	Contact entities are abstract entities that exist in the form of contact subtypes, such as Person and Doctor. The ContactAdded, ContactChanged, and ContactRemoved events are sent for all contact subtypes. Also, contrast Contact with ClaimContact and ClaimContactRole.
	PersonalDataPurge	A Contact purge was committed as part of a personal data purge. You must define your own rule to handle this event.
Doctor	ContactAdded ContactChanged ContactRemoved	Standard events for Contact entities.
LawFirm	ContactAdded ContactChanged ContactRemoved	Standard events for Contact entities.
LegalVenue	ContactAdded ContactChanged ContactRemoved	Standard events for Contact entities.
MedicalCareOrg	ContactAdded ContactChanged ContactRemoved	Standard events for Contact entities.
Organization	OrganizationAdded OrganizationChanged OrganizationRemoved	Standard events for Organization entities.
Person	ContactAdded ContactChanged ContactRemoved	Standard events for Contact entities.
VendorVendor	ContactAdded ContactChanged ContactRemoved	Standard events for Contact entities.
Place	ContactAdded ContactChanged ContactRemoved	Standard events for Contact entities.

Triggering a remove-related event

The *EntityNameRemoved* events trigger after either of the following occurs.

- Code deletes an entity from ClaimCenter
- Code marks the entity as retired. Retiring means a logical delete but leaves the entity record in the database. In other words, the row remains in the database and the Retired property changes to indicate that the entity data in that row is inactive. Only some entities in the data model are retireable. Refer to the *Data Dictionary* for details.

Triggering custom events

A business rule can trigger a custom event by using the `addEvent` method of claim entities and most other entities:

```
addEvent(strEventId : String) : void
```

You can also call the method from Java code that uses the Java API libraries, specifically from Java plugins or from Java classes called from Gosu:

```
void addEvent(String strEventId)
```

The `strEventId` can be any text you wish to use to identify the event. The `addEvent` method has no return value.

The entity that adds an event is the root object of that event. The event is fired when the entity's bundle is committed. The order in which added events are processed is not guaranteed.

Custom events can be implemented in a messaging plugin to acknowledge a message. They can also be used to respond to data changes initiated by a user or a web service. Finally, an event-based rule can be defined to encapsulate code for a particular operation. The event can be triggered in a rule set, such as validation or from PCF pages, and then handled in the Event Fired rules.

Custom events from SOAP acknowledgments

Integrations that use SOAP API to acknowledge a message can use a separate mechanism for triggering custom events as part of the message acknowledgment. This approach is a common one for some financial events.

First, your web service API client code in Java creates a new web service DTO called `Acknowledgement`. Next, call its `setCustomEvents` method to store a list of custom events to trigger as part of the acknowledgment. Pass an array of `CustomEvents` objects, each of which encapsulates the `String` name of the event.

Then, submit the acknowledgment by using the `MessagingToolsAPI` web service.

```
messagingToolsAPI.ackMessage(myAcknowledgement);
```

See also

- “Using web services to submit ACKs and errors from external systems” on page 399

How custom events affect pre-update and validation

Be aware that pre-update and validation rules do not run solely because of a triggered event. An entity's pre-update and validation rules run only if actual entity data changes. In cases where triggered events do not correspond to entities with modified properties, the event firing alone does not trigger pre-update and validation rules. This does not affect most events, since almost all events correspond to entity data changes.

However, for the `ClaimResync` event (triggered from a claim resync from the user interface), no entity data inherently changes due to this event, so this difference affects resync handling. This also affects any other custom event firing through the `addEvent` entity method. If you also use `ContactManager`, this also affects `ContactManager`'s entity validation for events such as the `ABResyncContact` event. If you require the pre-update and/or validation rules to run as part of custom events, you must modify some property on the entity or those rule sets do not run.

Events for contact changes

The `Contact` object gets special handling because it is so widely used and in very different ways by different destinations. Some external systems have a relational model for contacts – a separate contact table or database, to which other entities reference. Other systems use a hierarchical model where the contact information is part of the claim record. Typically in these systems, contact information appears throughout the claim data model and linked information.

Accordingly, ClaimCenter reports `Contact` changes in two ways. A system that uses a relational model for contacts only needs to know if a contact itself changes. It can then update its contact record, and all references to the contact point to the new information. In ClaimCenter, there are actually two levels at which contact information exists.

First, a single copy of a contact is on each claim. If the contact has multiple roles on the claim, the contact has a single record. For example, one contact might have more than one of the following roles: insured, witness, claimant, alternative contact for another claimant. This single `Contact` entity links to the claim by the claim contact (`ClaimContact`) entity. The claim contact entity contains multiple roles describing the different roles, which are `ClaimContactRole` entities.

Second, a contact may have a master record in an external address book (typically Guidewire ContactManager) shared across multiple claims (and even multiple systems).

An external system might want to know the following information.

- If the master record changed.
- If the contact snapshot associated with a single claim changed.
- If the roles played by the contact within the claim changed.

Each of these situations has a corresponding event in ClaimCenter.

A non-relational system needs additional context. It need to know not only what information on the contact changed, but what objects see the contact so that it can make updates to those objects internally. For example, suppose the same contact is a claimant on multiple exposures and the claimant address exists on each exposure record in the external system. In this case, a single change to a `Contact` entity in ClaimCenter requires updates to multiple exposure records in the external system.

Register only for the events that are appropriate for how each destination system works. You must understand how ClaimCenter and your external systems handle contact information.

The following examples show how ClaimCenter treats some contact changes.

- An adjuster adds a new witness to an existing exposure. She makes a new contact record for the witness. This triggers a `ContactAdded` event for the new person and a `ClaimContactAdded` event for the association of that person with the claim in the role of witness.
- The adjuster later realizes the witness was also a passenger, a custom role you might define. This triggers a `ClaimContactRoleAdded` event.
- The adjuster then updates the contact record for this person. This triggers a `ContactChanged` event for the shared contact record. This triggers a `ClaimContactContactChanged` event to alert you that the contact record changed for a person associated with a particular claim.
- The adjuster decides to promote this person to the master address book in case the person relates to future claims. This triggers an `ABCContactAdded` event.
- If it turns out the passenger is not a witness and you remove this role, ClaimCenter triggers the `ClaimContactRoleRemoved` event.
- If a contact is no longer the active person playing that role and the role is inactive, this raises a `ClaimContactRoleChanged` event. For example, this happens if the person was the primary doctor but the claimant switches doctors.
- If you change a person's information in the user interface and indicate changes in the central address book, the system sends two events. ClaimCenter triggers both `ContactChanged` and `ClaimContactContactChanged` events.

These examples lead to the following conclusions.

- Contact events are usually not useful. Multiple claims never share a contact. If you want to know that a contact on a claim changed, listen for the more specific `ClaimContactContactChanged` event. This event tells you the claim that it occurs on and which contact changed.
- There are only a few cases where an object directly links directly to a contact rather than through the `ClaimContact` mechanism. For example, lienholders on a vehicle or property link directly link to the contact, so these are the only places where a contact change would not trigger a `ClaimContactContactChanged` event. Otherwise, do not bother listening for `ContactChanged` events.

- For the most part, concentrate on checking `ClaimContact` and `ClaimContactRole` events for whoever is the claimant, insured, or lawyer. Check for `ClaimContactChanged` events for changes to the person's information or company's information.

Finally, some additional notes related to changes to contacts.

- If a claim adds a `ClaimContact`, ClaimCenter only triggers the `ClaimContactAdded` event. There are no separate events triggered for each role added as a consequence of adding the `ClaimContact`. This means that the messages that respond to the `ClaimContactAdded` event must look at all roles for the new `ClaimContact`.
- If some code removes a claim contact from a claim, ClaimCenter triggers the `ClaimContactRoleRemoved` event and then the `ClaimContactRemoved` event. The separate events triggered for each role help determine what roles the contact previously played so that these can be cleared in an external system. If the `ClaimContactRemoved` event triggers, the contact's roles are already gone. The `ClaimContactRoleRemoved` events provide an opportunity to determine what roles the contact used to have so that you can synchronize them with an external system.
- Policies and related objects (vehicles, properties) link to contacts. ClaimCenter reports `ContactChanged` events if the person's address book information changes. However, it does not report anything else for these references because it does not report changes to these objects anyway.
- The claimant information links to a contact in notes, documents, activities, evaluations, negotiations, and checks. ClaimCenter does not report changes to top-level objects such as these if the contact record changes. However, it does report a change if these objects relate to a different claimant, as it would for any other property that changes on the top-level object.
- Checks reference contacts using the `cc_checkPayee` join table. ClaimCenter does not report changes to a contact's information as a change to the check. This is because the Pay To line on the check derives from the contact but ClaimCenter saves this separately at the time it creates the check.

Events for special subobjects

For changes to some subobjects that are really just extensions of a parent object, ClaimCenter triggers a “changed” event on the parent object.

Subobject type	Can be part of this object
Addresses	claims (for example, loss location), contacts
Vehicles	claims, vehicle damage exposures, and/or a policy
Property	property claims, some exposure types, and/or a policy
Employment Data	workers' compensation claims
Lost Wages Benefits	lost wages exposures
Endorsements	a policy
Stat codes	a policy

Ordering events

ClaimCenter generates the events in the order registered by the destination. For each event name in the list, if one or more corresponding object changes occurred in the transaction, ClaimCenter generates an event for each distinct object change. For example, let's assume the registered event list includes the following events.

- `ClaimAdded`
- `ExposureAdded`
- `ClaimChanged`
- `ExposureChanged`

If you create an exposure and update two of its exposures in one operation, ClaimCenter generates three events:

- ExposureAdded
- ExposureChanged
- ExposureDeleted

No events from import tools

The web services interface `ImportToolsAPI` and the corresponding `import_tools` command line tool are a generic mechanism for loading system data or sample data into the system. Events do not trigger in response to data added or updated using this interface. Be very careful about using this interface for loading important business data where events might be expected for integration purposes. You must use some other system to ensure your external systems are up to date with this newly-loaded data.

Similarly, importing any data from staging tables with the `ITableImportAPI` web service does not trigger events.

Generating new messages in Event Fired rules

Each time a system event triggers a messaging event, ClaimCenter calls the Event Fired rule set. The application calls this rule set once for each event/destination pair for destinations that are interested in this event. Destinations signal which events they care about in the Messaging editor in Guidewire Studio™, which specifies your messaging plugins by name. The plugin name is the name for which Studio prompts you when you register a plugin in the Plugins editor in Studio. Your Event Fired rules must decide what to do in response to the event. Most importantly, decide whether you want to create a message in response to the event.

Message creation impacts user response times, so avoid unnecessarily large or complex messages. Event Fired rules run in the context of the transaction that changed entity data. If a user initiated the change, the processing is synchronous from the user perspective. Any resource-intensive or high-latency operations will degrade user interface responsiveness. Depending on the type of change, you can consider moving some resource-intensive operations to the *before-send* phase of message sending.

The most important object your Event Fired rules use is a message context object, which you can access by using the `messageContext` variable. This object contains information such as the event name and destination ID. Typically your rule set generates one or more messages, although the logic can omit creating messages as appropriate. You can use business rules to analyze the event and generate messages.

Studio includes a tool that helps you export business data entities and other types, like Gosu classes, to XML. You can select which properties are required or optional for each integration point. You can export an XSD to describe the data interchange format you selected. Then, you can edit your Event Fired rules to generate a payload for the entity that conforms to your custom XSD.

See also

- “[Restrictions on entity data in messaging rules and messaging plugins](#)” on page 380

Rule set structure

If you look at the sample Event Fired rule set, you can see a suggested hierarchy for your rules. The top level creates a different branch of the tree for each destination. You can determine which destination this event applies to by using the `messageContext` variable accessible from `EventFired` rule sets. For example, to check the destination ID number, use code like the following statement.

```
// If this is for destination #1  
messageContext.DestID == 1
```

At the next level in the rules hierarchy, it determines for what root object an event triggered.

```
messageContext.Root typeis Claim// If the root object is a Claim...
```

Finally, at the third level there is a rule for handling each event of interest.

```
messageContext.EventName == "ClaimChanged"
```

In this way, it is easy to organize the rules and keep the logic for handling any single event separate. Of course, if you have shared logic that would be useful to processing multiple events, create a Gosu class that encapsulates that logic. Your messaging code can call the shared logic from each rule that needs it.

Simple message payload

There are multiple steps in creating a message. First, you must cast the root object of the event to a variable of known type.

```
var claim = messageContext.Root as Claim
```

Once the Rule Engine recognizes the root object as a `Claim`, it allows you to access properties and methods on the claim to parameterize the payload of your message.

Next, create a message with a `String` payload.

```
var msg = messageContext.createMessage("The claim number is " + claim.ClaimNumber +
" and event name is " + messageContext.EventName)
```

Safe-ordering of messages

If you want to use the safe ordering feature of ClaimCenter, you may need additional lines of code, depending on what the root object of the event is.

Multiple messages for one event

The Event Fired rule set runs once for each event/destination pair. Therefore, if you need to send multiple messages, create multiple messages in the desired order in your Event Fired rules.

```
var msg1 = messageContext.createMessage("Message 1 for claim " + claim.PublicID +
" and event name " + messageContext.EventName)
var msg2 = messageContext.createMessage("Message 2 for claim " + claim.PublicID +
" and event name " + messageContext.EventName)
```

You can also use loops or queries as needed. For example, suppose that if a claim-related event occurs, you want to send a message for the claim and then a message for each note on the claim. The rule might look like the following code statements.

```
var claim = messageContext.Root as Claim
var msg = messageContext.createMessage("message for claim with public ID " + claim.PublicID)

for (note in claim.Notes) {
    msg = messageContext.createMessage(note.Body)
}
```

This creates one message for the claim and also one message for each note on the claim.

If you create multiple messages for one event like this, you can share information easily across all of the messages. For example, you could determine the username of the person who made the change, store that in a variable, and then include it in the message payload for all messages.

Remember that if multiple destinations requested notification for a specific event name, your Event Fired rule set runs once for each destination, varying only in the `messageContext.DestID`.

Determining what changed

In addition to normal access in a rule to the root object of a `messageContext` object, there is a way to find out what has changed. Your business rule logic can determine which user made the change, the time stamp, and the original value of changed properties. This information is available only at the time you originally generate the message, which is called

early binding. You cannot use the `messageContext` object during processing of late bound properties, which are properties that are bound immediately before sending.

At the beginning of your code, use the `isFieldChanged` method to test whether the property changed. If the field changed, and only if it changed, call the `getOriginalValue` method to get the original value of that property. To get the new (changed) value, access the property directly on an entity. The new value has not yet been committed to the database. There are additional methods similar to `isFieldChanged` and `getOriginalValue` that are useful for array properties and other situations.

For example, the following Event Fired rule code checks if a property changed and also checks its original value.

```
Var usr = User.util.getCurrentUser() as User
Var msg = "Current user is " + usr.Credential.UserName + "."
msg = msg + " current loss type value is " + claim.LossType
if (claim.isFieldChanged("LossType")) {
    msg = msg + " old value is " + (claim.getOriginalValue("LossType") as LossType).Code
}
}
```

Rule sets must never call message methods for ACK, error, or skip

From within rule sets, you must never call any message acknowledgment or skipping methods such as the `Message` methods `reportAck`, `reportError`, or `skip`. Use those `Message` methods only within messaging plugins. This prohibition also applies to Event Fired rules.

Save values across rule set executions

A single action in the user interface can generate multiple events that share some of the same information. Imagine that you do some calculation to determine the user's ID in the destination system and want to send this ID value in all messages. You cannot save the ID value in a variable in a rule and use it in another rule. The built-in scope of variables within the rule engine is a single rule. You cannot use the information later if the rule set runs again for another event caused by the same user interface action. ClaimCenter solves this problem by providing a `HashMap` that you can access across multiple rule set executions for the same action that triggered the system event.

Two API methods are available on the object returned by the Gosu expression `messageContext.SessionMarker`. Both methods create a hash map that exists for multiple Event Fired rules executing in a single database transaction triggered by the same system event. There is an important difference between the two methods.

- To write to a hash map that exists for the lifetime of all rules for all destinations, call the method `addToSessionMap(key, value)`. To read from the hash map, call the `getFromSessionMap(key)` method.
- To write to a hash map that exists for the lifetime of all rules for the current messaging destination only, call the method `addToTempMap(key, value)`. To read from the hash map, call the `getFromTempMap(key)` method.

For example, suppose that in a single action, an activity completes and it creates a new note. This change causes two different events and hence two separate executions of the `EventFired` rule set. As ClaimCenter executes rules for completing the activity, your rule logic could save the subject of the activity by adding it to the temporary map using the `SessionMarker.addToTempMap` method. Later, if the rule set executes for the new note, your code checks if the subject is in the `HashMap`. If it is in the map, your code adds the subject of the activity to the message for the note.

Code to save the activity's information would look like the following statements.

```
var session = messageContext.SessionMarker // get the sessionmarker
var act = messageContext.Root as Activity // get the activity

// Store the subject in the "temporary map" for later retrieval!
session.addToTempMap( "related_activity_subject", act.Subject )
```

Later, to retrieve stored information from the `HashMap`, your code would look like the following statements.

```
var session = messageContext.SessionMarker // get the sessionmarker

// Get the subject line from the "temporary map" stored earlier!
var subject = session.getFromTempMap("related_activity_subject") as String
```

If you need to add an entity instance to the bundle, explicitly add it to the bundle. Get the correct bundle using the Gosu expression `messageContext.Bundle`. Add entities to the bundle before adding it to the hash map.

```
var findResult = mycompany.QueryUtils.findRelatedObject().AtMostOneRow /* your own database query */
var resultToAdd = (findResult == null) ? null : messageContext.Bundle.add(findResult)
messageContext.SessionMarker.addToTempMap("MyKey", resultToAdd)
```

You can use this API to ensure that important Event Fired code not get run twice. Set data in the map, and later your code can check data in your map to see if it already ran.

Creating a payload by using Gosu templates

You can use Gosu code in business rules to generate Gosu strings using concatenation to design message payloads, which are the text body of a message. Generating your message payloads directly in Gosu offers more control over the logic flow for the messages you need and for using shared logic in Gosu classes.

However, sometimes it is simpler to use a text-based template to generate the message payload text. This is particularly true if the template contains far more static content than code to generate content. Also, templates are easier to write than constructing a long string by using concatenation with linefeed characters. Particularly for long templates, templates expose static message content in simple text files. People who might not be trained in Guidewire Studio or Gosu coding can easily edit these files.

You can use Gosu templates in business rules to create some or all of your message payload.

For example, suppose you create a template file `NotifyAdminTemplate.gst` in the `mycompany.templates` package. The fully-qualified name of the template is `mycompany.templates.NotifyAdminTemplate`.

Use the following code to generate a template and pass a parameter.

```
var myClaim = messageContext.Root as Claim;
// generate the template and pass a parameter to the template
var x = mycompany.templates.NotifyAdminTemplate.renderToString(myClaim)
// create the message
var msg = messageContext.createMessage("Test my template content: " + x)
```

The code assumes the template supports parameter passing. For example, as in the following invocations.

```
<%@ params(myClaimParameter : Claim) %>
The Claim Number is <%= myClaimParameter.ClaimNumber %>
```

There are several steps:

1. Select the template.
2. Allow templates to use objects from the template's Gosu context using the `template.addSymbol` method.
3. Execute the template and get a `String` result you could use as the message payload, or as part of the message payload.

The `addSymbol` method takes the symbol name that is available from within the template's Gosu code, an object type, and the actual object to pass to the template. The object type could be any intrinsic type, including ClaimCenter entities, such as `Claim` or even a Java class.

Setting a message root object or primary object

From the Gosu environment, the `messageContext.Root` property specifies the root object for an event. Typically, this same object also the root object for the message generated for that event. Because that is the most common case, by default any new message gets the same root object as the event object root. The message root indicates which object this message is about.

You can override the message root object to be a different object. For example, suppose you added a subobject and caught the related event in your Event Fired rules and added a message. You might want the message root to be the subobject's parent object instead of the default behavior. To override the message root, set the message's `MessageRoot` (not `Root`) property in your Event Fired rules.

```
message.MessageRoot = myObject
```

It is important to note that the primary object of a message is different from the message root, however. It is actually the primary object of a message that defines the message ordering algorithm.

Unlike the message root, there is not a single property that implements the primary object data for a message. Instead, there are multiple properties on a Message object that correspond to each type of data that could be a primary object for that application. Each application can define a default primary entity. Each messaging destination can choose from a small set of primary entities. The properties on the Message object for primary entity are strongly typed to the type of the primary entity. In other words, there is a different property on Message for each primary entity type.

In ClaimCenter, there are multiple properties on Message for the primary entity.

Property	Description
message.Claim	The claim, if any, associated with this Message object.
message.Contact	The contact, if any, associated with this Message object.

If you set the `message.MessageRoot` property, the following behavior occurs automatically as a side-effect of setting this property from Gosu or Java.

Condition	Behavior
If the entity is one of the following entity types:	ClaimCenter automatically sets the <code>message.Claim</code> property to set the primary entity.
<ul style="list-style-type: none"> • Claim • Policy • ClaimInfo • ClaimContactRole • Entity with a direct foreign key to Claim 	
If both of the following are true:	ClaimCenter automatically sets the <code>message.Contact</code> property to set the primary entity.
<ul style="list-style-type: none"> • The entity type is Contact • The primary object type is Contact 	

IMPORTANT: To ensure safe-ordering of messages, Guidewire recommends that you set the value of both `message.Claim` and `message.Contact` directly. This ensures that the messages for a particular entity type are safe-ordered on the root `Claim` or `Contact` of which the entity is part.

You only need to set the primary entity properties manually if the automatic behaviors described in this topic did not set them already. Note that you do not need to set unused primary entity properties to `null`. The only primary entity property used in the Message object is the one that matches the primary entity for the destination.

To configure the behavior of the message ordering system for safe ordering, perform the following actions.

1. Set the alternative primary entity in the messaging destination.
2. Set the root object to an entity from which you can determine the primary entity. For example, ClaimCenter can extract a Contact from either a Contact entity or ClaimContact entity.

Alternatively, you can set the column associated with that entity directly. Each candidate primary entity has a foreign key, indexed, column in the Message entity. If it is not possible to determine the appropriate entity object from the root object, you can explicitly set the appropriate foreign key column to the desired primary key object.

Be careful with setting message properties that store a reference to a primary object. ClaimCenter uses that information to implement safe ordering of messages by primary object.

Example

For example, suppose you want to use the default message root for an event that operates on a `Claim` object. However, you want to safe order the message on a destination with `Policy` as the primary entity. Your message creation code in Event Fired rules looks like the following statements.

```
ACTION (messageContext : entity.MessageContext, actions : gw.rules.Action)
var claim = messageContext.Root as Claim
var message = messageContext.createMessage(claim.DisplayName) // generate your payload somehow
message.Claim.Policy = claim.Policy // set a primary entity property
```

Now suppose you want to set the message root to a specific `Policy` object that is not the message root object. As with the previous example, assume the message is for a destination with `Claim` as the primary entity. Your message creation code in Event Fired rules looks like the following statements.

```
ACTION (messageContext : entity.MessageContext, actions : gw.rules.Action)
var claim : Claim // some specific claim
var message = messageContext.createMessage(claim.DisplayName) // generate your payload somehow
message.MessageRoot = claim
message.Claim.Policy = claim.Policy
```

See also

- “[Sending safe-ordered messages](#)” on page 352

Creating XML payloads by using GX models

Guidewire Studio provides a tool that helps you export business data entities and other types like Gosu classes to XML. You can select whether properties are required or optional for each integration point. You can export an XSD to describe the data interchange format you selected. Then, you can use this model to export XML from or import XML into your integrations. For example, your messaging plugins or your Event Fired rules could send XML to external systems. You could also write web services that take XML data payloads from an external system or return XML as the result.

Using Java code to generate messages

Business rules, including message-generation rules, can optionally call out to Java modules to generate the message payload string.

Saving attributes of a message

As part of creating a message, you can save a message code property within the message to help categorize the types of messages that you send. Optionally, you can use this information to help your messaging plugins handle the message. Additionally, your destination could report on how many messages of each type were processed by ClaimCenter, such as for reconciliation.

If you need additional properties on the `Message` entity for messaging-specific data, extend the data model with new properties. Only do this extension for messaging-specific data.

During the `send` method of your message transport plugin, you could test any of these properties to determine how to handle the message. As you acknowledge the message, you could compare values on these properties to values returned from the remote system to detect possible mismatches.

ClaimCenter also lets you save entities by name, saving references to objects with the message to update ClaimCenter entities as you process acknowledgments. For example, to save a `Note` entity by the name `note1` to update a property on it later, use code similar to the following statement:

```
msg.putEntityByName("note1", note)
```

These methods are especially helpful for handling special actions in acknowledgments. For example, to update properties on an entity, use these methods to authoritatively find the original entity. These methods work even if public IDs or other properties on the entity change. This approach is particularly useful if public ID values could change

between the time Event Fired rules create the message and the time your messaging plugins acknowledge the message. The `getEntityByName` method always returns the correct reference to the original entity.

Handling policy changes on a claim

ClaimCenter stores a snapshot of policy information for a claim. However, you can enter or edit policy information in ClaimCenter if no policy system integrates with ClaimCenter. In this case, you may need to send changes made to the policy with claim information to an external system. ClaimCenter generates policy and coverage-related events to help you generate messages if the policy information changes.

Multiple claims never share a policy snapshot so the `Policy.claims` array only ever contains one claim.

Maximum message size

Messages can contain up to one billion characters.

If multiple events fire, which message sends first?

For each destination, the Event Fired rules run in the order that each destination's configuration specifies in the Messaging editor in Studio. In general, messages send in the order of message creation in Event Fired rules.

ClaimCenter runs the Event Fired for one event name before the rules run again for the next listed event name. For messages with both the same event name and same destination, the message order is the order that your Event Fired rules create the messages.

In typical deployments, this means that the event name order in the destination setup is very important. Carefully choose the order of the event names in the destination setup. It is important to remember that changes to the ordering of the event names change the order of the events in Event Fired rules. Such changes can produce radical effects in the behavior of Event Fired rules if they assume a certain event order. For example, typical downstream systems want information about a parent object before information about the child objects.

The event name order in the destination setup is critical. Carefully choose the order of the event names in the destination setup. Be extremely careful about any changes to ordering event names in the destination setup. Changes in the event name order could change message order, and that can force major changes in your Event Fired rules logic.

Because ClaimCenter supports safe-ordering of messages related to a primary object, the actual ordering algorithm is more complex.

Restrictions on entity data in messaging rules and messaging plugins

Event Fired rules and messaging plugin implementations have limitations about changing entity instance data. Messaging code in these locations must perform only the minimal data changes necessary for integration on the message entity.

Event fired rule set restrictions for entity data changes

Entity changes in Event Fired rule sets must be very limited. The restrictions listed below apply to all entity types, including custom types. Design your messaging rules carefully around these restrictions to avoid data corruption and logical errors that are difficult to diagnose.

- In general, perform any data updates in pre-update rules rather than in Event Fired rules.
- A property is safe to change in Event Fired rules only if it is messaging-specific and users can never modify it from the user interface, even indirectly. Carefully consider the role of every property you might modify.
- The only object that is safe to add is a new message using the `createMessage` method of the `MessageContext` object. Never create new objects of any other type, even indirectly through other APIs.

- Never delete objects.
- Never call business logic APIs that might change entity data, even in edge cases.
- Never rely on any entity data changes triggering the following common rule sets.
 - Pre-update rule set
 - Validation rule set
 - Event Fired rule set

Messaging plugin restrictions for entity data changes

Entity changes in messaging plugin code must be very limited. The restrictions listed below apply to code triggered by a `MessageTransport` or `MessageReply` plugin implementation. Design your messaging code carefully around these restrictions to avoid data corruption and logical errors that are difficult to diagnose.

- A property is safe to change only if it is messaging-specific and users can never modify it from the user interface, even indirectly. Carefully consider the role of every property you might modify.
- With very few exceptions, it is not possible to create new objects messaging plugin code. In the default configuration, only the following objects are safe to create within a messaging plugin:
 - `Activity` objects
 - `Note` objects
 - `Workflow` objects
 - `WorkQueue` and `WorkItem` objects
 - `OutboundRecord` objects

You cannot rely on pre-update rules or validation rules running for those objects.

All other object types are dangerous and unsupported to add from within messaging plugins.

If you modify the data model such that there are additional foreign keys on these objects, even these objects explicitly listed may be unsafe to add.

- Never delete objects.
- You must not rely on any entity data changes eventually triggering the following common rule sets.
 - Pre-update rule set
 - Validation rule set
 - Event Fired rule set for the standard events `ENTITYAdded`, `ENTITYChanged`, and `ENTITYRemoved`
However, an Event Fired rule set is still triggered for events that you explicitly add. You can use the API `entity.addEvent` method to add events. ClaimCenter calls the Event Fired rule set once for each custom event for each messaging destination that listens for it.
- Never call business logic APIs that might change entity data, even in edge cases.
- From messaging plugin code, entity instance changes do not trigger concurrent data exceptions except in special rare cases. To avoid data integrity issues with concurrent changes, avoid changing data in these code locations.
- In some cases, consider adding or advancing a workflow as an alternative to direct data modifications from messaging code. The workflow can asynchronously perform code changes in a separate bundle outside your messaging-specific code.
- If you must update messaging-specific data, consider how absence of detecting concurrent data changes from messaging plugins might affect your extensions to the data model. For example, suppose you intend to modify an entity type to add a property with simple data. Instead, you could add a property with a foreign key to an instance of a custom entity type. First, create the instance of your custom entity type at an early part of the lifecycle of your main objects before the messaging code runs. As mentioned earlier, it is unsupported to create an entity instance in Event Fired rules or in messaging plugins. This restriction applies to all entity types, including custom entity types.

In Event Fired rules or in messaging plugins, modify the messaging-specific entity instance. With this design, there is less chance of concurrent data change conflicts from a simple change on the main business entity instance from within the user interface.

There also exists an optional feature to lock related objects during messaging actions. With locked data—usually the primary entity instance or the message—any attempt to access the locked data causes the accessing code to wait until the data is unlocked. For maximum data integrity, enable entity locking during messaging. For maximum performance, disable entity locking during messaging.

Database transactions when creating messages

A single database transaction comprises all the steps up to and including the adding of messages to the send queue. The database transaction is always the same transaction that triggers the initial event.

If any of the following exceptions or errors occur, the database transaction rolls back, including all messages added to the send queue.

- Exceptions in rule sets that run before message creation
- Exceptions in Event Fired rules (where you create your messages)
- Exceptions in rule sets that run after Event Fired rules but before committing the bundle to the database
- Errors committing the bundle to the database, and remember that this bundle includes new Message objects

Messaging plugins must not call SOAP APIs on the same server

In general, avoid calling locally-hosted SOAP APIs from within a plugin or the rules engine. There are various problems if you call SOAP APIs that modify entities that are currently in use, including but not limited to APIs that might change the message root entity. Be careful about any SOAP calls to the same server. If the SOAP API hosted locally modifies entity data and commits the bundle, the current transaction does not always detect and reload local data.

Instead, refactor your code to avoid this case. For example, write a Gosu class that performs a similar function as the web service but that does not commit the bundle. This type of refactoring also results in higher server performance.

This is true for all types of local loop-back SOAP calls to the same server.

Those limitations are true for all plugin code. In addition, there are messaging-specific limitations with this approach. Specifically, ClaimCenter locks the root entity for the message in the database. Any attempts to modify this entity from outside your messaging plugin (and SOAP APIs are included) result in concurrent data exceptions.

Delaying or censoring message generation

Detect claim state changes, such as from draft to open

Your event rules in the Event Fired rule set can detect a claim state change using the `isFieldChange` and `getOriginalValue` methods.

For example, you can detect the one-time state change from the state `draft` to the state `open`. Check for the following condition.

```
claim.isFieldChange("State")
and claim.State == ClaimState.TC_OPEN
and claim.getOriginalValue("State") == ClaimState.TC_DRAFT
```

Similarly, you can check whether `claim.State` is `TC_DRAFT`, and if so then discard most messages.

Validity and rule-based event filtering

ClaimCenter allows entry and creation of claims with fewer restrictions than might be true for an external system, such as a mainframe. ClaimCenter does this so that a new claim can be committed to the database with minimal information and the claim can be improved later as you gather more information.

However, in some cases, an external system might not want to know about the claim until a much more complete (or perhaps more correct) claim commits to the database. You might choose to not send a claim added message to the mainframe if there is not enough information to create the record on the mainframe. Express this level of correctness or completeness by setting a validation level for the claim. Each external system's destination defined in ClaimCenter may have completely different validation requirements.

For example, suppose an external system wants to be notified of every claim, but its standards were high about what kinds of claims it wants to know about. Additionally, you might want to omit notifying an external system about new notes on a claim that it does not know about yet. In other words, you did not send a message notifying the external system about the claim, so you probably do not want to send updates about its subobjects.

To implement this, there are two parts of the integration implementation.

1. Validation rules implicitly set the validation level – There are two rule sets that govern validation checking, one for claims (Claim Validation Rules) and one for exposures (Exposure Validation Rules). These rules can set the validation level. The validation rules engine sets the `claim.ValidationLevel` property to the highest level that does not have rule rejection messages.
2. Your event rules check validation level (and other settings) – Your Event Fired rules define whether to send a message to an external system. For example, the rule might listen for the events `ExposureAdded`, `ExposureChanged`, `ClaimAdded`, `ClaimChanged`, and perhaps other events. If the event name and destination ID matches what it listens for and the validation level was greater than a certain level, the rule creates a new message. A sample Gosu rule is shown below.

```
if (claim.ValidationLevel > externalSystemABCLevel) {  
    // create message...  
}
```

It is important to understand that you define your own standards for event filtering and processing. ClaimCenter does not enforce any requirements about validation levels. The `EntitynameAdded` events and `EntitynameChanged` events trigger independent of the object's validation level.

Generally speaking, ClaimCenter does not use the validation levels. You can define rules that set or get the validation level for some purpose. Your messaging rules might send a message to an external system because of an event, but ignore the event in some cases due to the validation rules.

There is one rule set that governs claim checking, called Claim Validation Rules. You can set validation levels in these rules that are checked later within the Event Fired rule set.

ClaimCenter itself does not actually use the validation level for any purpose. However, if you use the ISO integration, the validation level defines whether the claim or exposure is ready for ISO.

Late binding data in your payload

In your Event Fired messages, in general it is best to use the current state of entity data to create the message payload. In other words, generate the entire payload when the Event Fired rule set runs. For example, for `ClaimChanged` events, messages typically contain the latest information for the claim as of the time the messaging event triggers. This information includes any changes in this database transaction, such as entity instance additions, removals, or changes.

If you wait until messaging sending time to create the message, the data can be partially different, or it might even have been removed from the database. These data changes can disrupt the series of messages to a downstream system. Downstream systems typically need messages that match with data model changes as they happen. Creating the entire payload at Event Fired time, called *early binding*, is the standard recommended approach.

An issue with early binding is that later changes to an object cannot be added to an earlier message about that object, especially if there is a large delay in sending the message. Sometimes you need the latest possible value on an entity as

the message leaves the send queue on its way to the destination. For example, you send a new claim to an external system. As part of the acknowledgment, the external system might send back its ID for the new claim. You can set the public ID in ClaimCenter to that external system's ID for the claim.

Suppose the next message sends a *reserve* for that claim to the external system. In this message, you include the new public ID for the claim, which was received from the external system in the acknowledgment. The external system can identify which claim belongs with this second message. If the public ID were merely set during the original processing of the event, the reserve message could not contain the new value from the external system. There would be no way to tell the external system which claim this second set of information goes with.

ClaimCenter solves this problem by permitting late binding of properties in the message payload. You can designate certain properties for late binding so you re-calculate values immediately before the messaging transport sends the message.

For typical situations, export all data in the Event Fired rules into a message payload, which is standard early-bound messaging. Use late binding when late binding is important for performance or if some payload data is not yet available in Event Fired rules.

For newly created entity instances, you can send the entity instance's public ID property as a late-bound property. A message acknowledgment or external system using web service APIs could change the public ID between creating the message and sending it.

For other properties, decide whether early binding or late binding is most appropriate. Even for late binding, there are several implementation options.

See also

- “Implement late binding” on page 384

Implement late binding

About this task

The following procedure assumes that there is only one token to transform for late binding. If you have multiple items that require late binding, repeat the procedure as appropriate.

Procedure

1. At message creation time in Event Fired rules, add your own marker text within the message. A simple example would be the arbitrary token <AAAAAA>.
2. Decide whether to implement late binding at before-send time or message-send time.
 - To minimize the latency time between message creation time, implement late binding at before-send time and enable Distributed Request Processing for that destination. The before-send processing happens as a distributed background task across potentially multiple nodes in your cluster even for a single destination.
If you enable Distributed Request Processing for the destination, message order for before-send processing is non-deterministic. Do not rely on a specific order for messages in your `beforeSend` code. For some use cases, this is an acceptable trade-off for performance. In other use cases, a late binding transformation cannot occur until all other messages for that primary object are already processed.
 - To ensure strict ordering with larger latency between message creation time and message send time, there are two choices. You can implement late binding at before-send time and do not enable Distributed Request Processing. If you need to enable Distributed Request Processing for other reasons, you can implement late binding in your `MessageTransport` plugin, although that is not generally recommended.
3. Add new code in the right location.
 - To implement late binding at before-send time, there are potentially two choices depending on how you configured your destination. You can implement a `MessageRequest` plugin, which also handles after send processing. Alternatively, implement a `MessageBeforeSend` plugin, which handles only before-send processing but requires that you enable Distributed Request Processing. In either case, your plugin

implementation must have a `beforeSend` method that takes a `Message` object and returns a `String` object that is the transformed payload.

- To implement late binding at send time, add code to your `MessageTransport` plugin implementation in the `send` method.
4. In your new late binding code, find the special token and then substitute or transform it. Depending on the context, you might need to get objects from the `Message` to check the current value. For example, perhaps you might get the `Message.MessageRoot` object and cast it to a `Claim`. Get whatever properties you need from it. The current value of the property is a late bound value. Replace the marker with the new value.

For example, a simple Gosu implementation of the `MessageRequest` or `MessageBeforeSend` plugin interface might use the following code in the `beforeSend` method. In this example, the transport assumes the message root object is a `Claim` and replaces the special marker in the payload with the value of extension property `SomeProperty`. This example assumes that the message contains the string `<AAAAAA>` as a special marker in the message text.

```
function beforeSend(m : Message) {  
    var c = m.MessageRoot as Claim  
    var s = org.apache.commons.lang.StringUtils.replace(m.getPayload(), "<AAAAAA>", c.SomeProperty)  
    return s  
}
```

For more complex substitutions, there are APIs that can search for tokens with special delimiters around them. You can provide a class that map any input token to a different output token.

See also

- “Map message payloads by using tokens” on page 394

Tracking a specific entity with a message

You can track a specific entity at message creation time in your Event Fired rules. You can use this entity in your messaging plugins during sending or while handling message acknowledgments. To attach an entity to a message in Event Fired rules, use the `Message` method `putEntityByName`. This method attaches an entity to this message and associates it with a custom ID called a *name*. Later, as you process an acknowledgment, use the `Message` method `getEntityByName` to find that entity attached to this message.

The `putEntityByName` and `getEntityByName` methods are helpful for handling special actions in an acknowledgments. For example, if you want to update properties on a certain entity, these methods authoritatively find the original entity that triggered the event. These methods work even if the entity’s public ID or other properties change. If the public ID on an object changes between the time of message creation and the time the messaging code acknowledges the message, `getEntityByName` always returns the correct entity.

For example, Event Fired rules could store a reference to an object with the name `abc:expo1`. In the acknowledgment, the destination would set the `publicID` property. For example, set the public ID of object `abc:expo1` to the value `abc:123-45-4756:01` to provide an ID for the external system.

Saving this name is convenient because, in some cases, the external system’s name for the object in the response is known in advance. You do not need to store the object type and public ID in the message to refer back to the object in the acknowledgment.

Implementing message plugins

Ensuring thread safety in messaging plugin code

All messaging plugin implementation code must be thread-safe. Be extremely careful about static variables and other shared memory structures. Multiple threads might access these items when running the same or related code.

Messaging plugin code must be thread-safe even if the Messaging editor **Number Sender Threads** field is set to 1.

Initializing a messaging plugin

To initialize a messaging plugin, the following plugin methods are called.

- **construct** – Constructor called when the object is created
- **setParameters** – Called to access parameters defined in the Guidewire Studio Plugins registry. This method is available to plugins that implement the `InitializablePlugin` interface.
- **setDestinationID** – Called to store the plugin's unique destination ID

The initialization methods set up the plugin for its subsequent operations. The methods must not throw an exception. If exception-throwing initialization code is required then best practice locates such code in the following methods.

- For plugins that implement the `MessageTransport` interface, place the initialization code at the beginning of the `send` method. The method can determine whether the plugin has already been initialized in an earlier call and, if needed, perform the appropriate setup operations.
- For plugins that implement the `MessageReply` interface, place the initialization code in the `initTools` method.

Getting messaging plugin parameters from the plugin registry

It may be useful in some cases to get parameters from the plugin registry in Studio. The benefit of setting parameters for messaging transports is that you can separate out variable or environment-specific data from your code in your plugin.

For example, you could use the Plugins editor in Studio for each messaging plugin to specify the following types of data for the transport.

- External system's server name
- External system's port number
- A timeout value

If you want to get parameters from the plugin registry, your messaging plugin must explicitly implement `InitializablePlugin` in the class definition. This interface tells the application that you support the `setParameters` method to get parameters from the plugin registry. Your code gets the parameters as name/value pairs of `String` values in a `java.util.Map` object.

For example, suppose your plugin implementation's first line looks like the following statement.

```
class MyTransport implements MessageTransport {
```

Add the `InitializablePlugin` interface to the definition.

```
class MyTransport implements MessageTransport, InitializablePlugin {
```

To conform to the new interface, add a `setParameters` method with the following signature.

```
function setParameters(map: java.util.Map) { // this is part of InitializablePlugin interface
    // access values in the MAP to get parameters defined in plugin registry in Studio
    var myValueFromTheMap = map["servername"]
}
```

Implementing message payload transformations before send

A destination can optionally define code that transforms the payload before sending. This is known as before-send processing.

There are a couple of situations in which before-send processing is desirable.

- Implementing late binding, which is a technique to substitute or insert data at message send time that was not available at message creation time.

- Implementing processing that is too complex and resource intensive to do at message creation time. For example, this code might take simple name/value pairs in the message payload and construct a large complex XML message in a format required by your messaging transport plugin.

By default, messaging plugins run only on the one server that handles this messaging destination. If more than one server qualifies for handling the destination's messaging operations, the cluster grants a lease to a single server. One server may handle multiple destinations, but, by default, one destination is handled by only a single server.

You can optionally share code across messaging destinations so they use the same message payload transformation code in their before-send processing.

You can register multiple implementations for each messaging plugin interface to communicate with multiple external systems. To distinguish them, as you create the plugin in Studio, Studio prompts you for a name for the plugin. That is called the plugin name. Use the plugin name when you configure plugins for the messaging destination in the Messaging editor in Studio.

Distributed processing

You can optionally enable Distributed Request Processing for the message destination, which permits multiple servers in the cluster to distribute the work for payload transformation for this destination. The Distributed Request Processing feature for a destination affects only before-send processing, not messaging sending in the transport, after-send processing, or asynchronous reply handling.

If you do enable Distributed Request Processing for the destination, do not rely on a message specific order in your messaging code in `beforeSend`. In this case, message order is non-deterministic for message request processing (your `beforeSend` method). The warning about non-deterministic order only applies to your `beforeSend` method, not the `send` method in your message transport plugin. Distributed Request Processing does not change send order for the `send` method, such as enforcing safe-ordered messaging, Strict Mode, and related settings. For some use cases, this is an acceptable trade-off for performance, but carefully consider your situation. If you require strict ordering, either disable Distributed Request Processing, or handle any transformations directly in your `MessageTransport` implementation. If you are doing the transformations to implement late-binding of data, it is possible that strict ordering is required, depending on your situation.

Implementing the `beforeSend` method

Implement your code in a `beforeSend` method to do the following:

- Take a `Message` object as input.
- Extract the `Message.Payload` property.
- Return a transformed payload as a `String` value.

You can implement this method in the following ways.

Without distributed request processing	Write a <code>MessageRequest</code> plugin implementation that contains the <code>beforeSend</code> method. This plugin must handle processing before sending in the <code>beforeSend</code> method and also after sending in the <code>afterSend</code> method. The <code>afterSend</code> method takes a <code>Message</code> entity instance and returns nothing. In the Messaging editor, register the plugin name in the Request Plugin field.
With distributed request processing	Write a <code>MessageBeforeSend</code> plugin implementation to use instead. The only required method is <code>beforeSend</code> . In the Messaging editor, register it in the Before Send Plugin field, in which case the destination ignores the Request Plugin field.

IMPORTANT: Do not call the `reportError` method in your implementation of the `beforeSend` method if using distributed message request processing.

The method return object

Your `beforeSend` method must return the transformed payload as a `String`. Do not modify the `Message.Payload` property directly. You can set other properties, including data model extension properties, on the `Message` entity instance. Any changes are persisted, assuming there are no exceptions thrown within the `beforeSend` method.

The application saves the transformed payload result of your `beforeSend` method so it is available to the messaging transport (`MessageTransport`) plugin in its `send` method as the `transformedPayload` parameter. If you implemented before-send processing, by default the transformed payload parameter contains a different value from the `Message.Payload` property. If for that destination, you selected both **Distributed Request Processing** and **Persist Transformed Payload**, `Message.Payload` has the same value as `transformedPayload`.

By default, this task runs only on the server that handles this messaging destination. If you enable Distributed Request Processing for the destination, the task is distributed across multiple servers in the cluster. In the Messaging editor, enable this feature by selecting the **Distributed Request Processing** checkbox.

Method exceptions

If your `beforeSend` method throws exceptions, the schedule for retrying is handled at send time by the standard destination settings for errors, such as the Retry Backoff Multiplier. This is true even if Distributed Request Processing is enabled.

Idempotent

You must ensure that your `beforeSend` implementation is idempotent, which means that it could be called multiple times and have the same effect. Your code must ensure that repeated calls cause the same results on the `Message` entity instance and the return value of the method. This warning applies to `beforeSend` methods implemented in either `MessageRequest` or `MessageBeforeSend` plugin interfaces. This warning applies independent of whether you use Distributed Request Processing. If the application attempts to retry sending a message, the application calls your `beforeSend` implementation more than once.

Message retry

The `Message` property called `Bound` is a boolean flag that specifies whether the message request processing is complete. You can optionally set `Bound` to `true` after message creation if you know that the message does not need processing. After the server performs the before-send processing by calling the appropriate `beforeSend` method, the server sets `Message.Bound` to `true`.

In message retry, ClaimCenter calls the `beforeSend` method again. To correctly handle message retry, your implementation of the `beforeSend` method must check the value of `Message.Bound` and behave accordingly as the message can have already set the value of the `Bound` property to `true`.

Implementing a message transport plugin

A destination must define a `MessageTransport` plugin to send a `Message` object over some physical or abstract transport. Sending the object might involve submitting a message to a message queue, calling a remote web service API, or implementing a complex proprietary protocol specific to some remote system. The message transport plugin is the only required plugin interface for a destination.

You can register multiple implementation for this interface to communicate with multiple external systems. To distinguish them, as you create the plugin in Guidewire Studio, Studio prompts you for a name for the plugin, called the *plugin name*. Use the plugin name when you configure the messaging destination in the separate Messaging editor in Studio. In the Messaging editor, for a destination, put the plugin name for your `MessageTransport` implementation in the **Transport Plugin** field.

To send a message, implement the `send` method, which ClaimCenter calls with a `Message` entity instance argument. The original payload is in the `Message.Payload` property. The `send` method has another argument, which is the transformed payload if that destination implemented a `MessageRequest` plugin.

In a message transport plugin's simplest form, the `send` method does its work synchronously entirely in the `send` method. For example, call a single remote API call such as an outgoing web service request on a legacy computer. For synchronous use, your `send` method immediately acknowledges the message with the code `message.reportAck(...)`. If there are errors, instead use the message method `reportError`. To report a duplicate message, use `reportDuplicate`, which is a method not on the current message but on the `MessageHistory` entity that represents the original message.

If you must acknowledge the message synchronously, your `send` method can optionally update properties on ClaimCenter objects, such as `Claim`. You can get the message root object by getting the property `theMessage.MessageRoot`. Changes to the message and any other modified entities persist to the database after the `send` method completes. Changes also persist after the `MessageRequest` plugin completes work in its `afterSend` method.

If your message transport plugin `send` method does not synchronously acknowledge the message before returning, then this destination must also implement the message reply plugin. The message reply plugin implementation must handle the asynchronous reply.

You must handle the possibility of receiving duplicate message notifications from your external systems. Usually, the receiving system detects duplicate messages by tracking the message ID, and returns an appropriate status message. The plugin code that receives the reply messages can call the `message.reportDuplicate` method. Depending on the implementation, the code that receives the reply would be either the message transport plugin or the message reply plugin. Your code that detects the duplicate must skip further processing or acknowledgment for that message.

If you want your plugin to get parameters from the plugin registry, implement the `InitializablePlugin` interface in the class definition. Next, add a `setParameters` method that gets the parameters as a `java.util.Map` object.

The following example demonstrates a minimal message transport in Gosu for testing.

```
uses java.util.Map;
uses java.plugin;

class MyTransport implements MessageTransport, InitializablePlugin {

    function setParameters(map: java.util.Map) { // this is part of InitializablePlugin
        // access values in the MAP to get parameters defined in plugin registry in Studio
    }

    // NEXT, define all your other methods required by the MAIN interface you are implementing...
    function suspend() {}

    function shutdown() {}

    function setDestinationID(id:int) {}

    function resume() {}

    function send(message:entity.Message, transformedPayload:String) {
        print("====")
        print(message.Payload) // the payload in the Message entity instance
        print("====")
        print(transformedPayload) // the payload set by the destination's MessageRequest plugin
        message.reportAck()
    }
}
```

For Java examples, see the following ZIP archive:

[ClaimCenter/java-examples.zip](#)

Implementing post-send processing

A destination can optionally define code that performs post-processing on the `Message` object immediately after sending the message with the `send` method of the `MessageTransport` plugin.

Implement your code in an `afterSend` method that takes a `Message` object and returns nothing. You can implement this in two ways.

- You can write a `MessageRequest` plugin implementation that contains this method. This plugin must handle processing before sending in the `beforeSend` method and also after sending in the `afterSend` method. In the Messaging editor, register the plugin name in the **Request Plugin** field.
- You can write a `MessageAfterSend` plugin implementation to use instead. The only necessary method is `afterSend`. In the Messaging editor, register it in the **After Send Plugin** field, in which case the destination ignores the **Request Plugin** field.

You can register multiple implementation for this interface to communicate with multiple external systems. To distinguish them, as you create the plugin in Studio, Studio prompts you for a name for the plugin. That is called the plugin name. Use the plugin name when you configure plugins for the messaging destination in the Messaging editor in Studio.

ClaimCenter calls the `afterSend` method immediately after the transport plugin's `send` method completes. If you implement asynchronous callbacks with a message reply plugin, it is critical to note the following situations.

- The server calls the `afterSend` method in the same thread (and database transaction) that runs the transport plugin's `send` method.
- The call to `afterSend` might be a separate thread and database transaction from any asynchronous reply callback code.

If no exceptions were thrown during the `MessageTransport` plugin `send` method, ClaimCenter calls the `afterSend` method in your `MessageAfterSend` or `MessageRequest` implementation. If the `send` method acknowledges the message reply synchronously by calling methods on the `Message`, it does not affect whether `afterSend` is called on your plugin implementation.

Implementing a MessageReply plugin

A message destination can asynchronously acknowledge a message by implementing the `MessageReply` plugin interface. As an example, the plugin might implement a trigger from an external system that notifies ClaimCenter whether a message-send operation succeeded or failed. Message acknowledgment is optional. In addition to acknowledging messages, a `MessageReply` plugin can perform other operations.

Multiple implementations of the `MessageReply` interface can be registered to communicate with multiple external systems. Each implementation has a unique name specified in the Guidewire Studio Plugin Registry **Name** field. When configuring plugins for the message destination in the Studio Messaging editor, reference the plugin name in the **Reply Plugin** field.

By default, the `MessageReply` plugin code runs on the server assigned to the message destination. If the plugin class is defined with the `@Distributed` annotation, the plugin code can run on any server in the cluster.

```
@Distributed  
public class MyMsgReplyPlugin implements MessageReply { ... }
```

The `MessageReply` plugin interface defines a single method called `initTools`. The method is called by the messaging infrastructure to initialize the plugin.

```
initTools(handler : PluginCallbackHandler, msgFinder : MessageFinder)
```

The `handler` argument provides a transaction context for executing the plugin code. The transaction context executes as the system user and includes a bundle for committing any changes performed by the plugin code.

The `msgFinder` argument can be used by the plugin code to look up the relevant message. The `MessageFinder` object provides methods such as `findById` and `findByRefId` which accept a `messageID` or `senderRefID`, respectively, and return the specified `Message` object.

The `initTools` method must save its arguments in private variables for subsequent use. The method has no return value.

Implement a PluginCallbackHandler

The plugin code that is executed in the context of a transaction is of type `PluginCallbackHandler.Block`. The `Block` is an interface that defines a single method called `run`.

```
run()
```

The `PluginCallbackHandler` interface also defines a method called `execute` which executes the code block passed to it.

```
execute(block : Block)
```

A `MessageReply` plugin method creates a new instance of the `PluginCallbackHandler.Block` interface that implements the `run` method. It then passes the `Block` to the `execute` method of the `PluginCallbackHandler` that was saved in the `initTools` method.

Alternatively, the plugin code can be contained in a Gosu anonymous function block that is passed directly to the `execute` method.

The `PluginCallbackHandler` interface also defines an `add` method to add an object to the transaction's bundle.

```
add(bean : Object) : Object
```

The `bean` argument references the object to add to the transaction's bundle. Changes to the object are committed to the database when the transaction completes.

The method returns a clone of the `bean` argument. All subsequent references to the object must be to this returned clone. All changes must also be performed on the clone. The original `bean` argument passed to the `add` method can be discarded.

The `add` method can be called only from within the `Block` interface's `run` method or its alternative implementation as a Gosu block. If the method is called from anywhere else, an exception is thrown.

In the following implementation of a simple `MessageReply` plugin method, the method skips processing the message that triggered the plugin code. A `Message` object provides additional methods, including acknowledging the message (`reportAck` method), re-sending the message (`retry` method), and reporting an error (`reportError` method). The example `MessageReply` plugin can be expanded to perform these operations by adding methods similar to the `skip` method.

The example code demonstrates the preferred manner of implementing a `MessageReply` plugin method. First, the plugin operations are exposed in their own interface. In the example code, the `skip` method is defined in the `IMyMessageReplyPlugin` interface.

```
public interface IMyMessageReplyPlugin extends gw.plugin.messaging.MessageReply {  
    public function skip(msgId : long) : void  
}
```

Then the exposed methods, plus the `initTools` method, are implemented in the plugin class. The example code implements the plugin in the `MyMessageReplyPlugin` class.

The `MessageReply` interface extends the `MessagePlugin` interface, which defines methods to control the messaging lifecycle, such as `suspend` and `resume`. The required `MessagePlugin` methods are implemented in the example code as empty methods.

Finally, the `skip` method demonstrates the two techniques for implementing and executing plugin code by defining either a Gosu block or a `Block.run` method.

```
class MyMessageReplyPlugin implements IMyMessageReplyPlugin {  
  
    // Saved initTools() handler and msgFinder arguments  
    private var _callbackHandler : gw.plugin.PluginCallbackHandler  
    private var _messageFinder : gw.plugin.messaging.MessageFinder  
  
    // Implement the MessageReply interface  
    override function initTools(handler : PluginCallbackHandler, msgFinder : MessageFinder) : void {  
        // Save the arguments for later use  
        _callbackHandler = handler  
        _messageFinder = msgFinder  
    }  
  
    // Methods defined in MessagePlugin interface (which MessageReply extends)  
    override function suspend() {}  
    override function resume() {}  
    override function shutdown() {}  
    override function setDestinationID(i : int) {}
```

```

// Implement the plugin skip operation exposed in IMyMessageReplyPlugin
override public function skip(msgId : long) {

    // Implement Block.run() method as a Gosu block/anonymous function, passing it to handler.execute
    _callbackHandler.execute( \ -> {
        // Use the saved initTools() msgFinder argument to retrieve the message
        var message = _messageFinder.findById(msgId)
        if (message != null) {
            message.skip()
        }
    })

    /* Alternative implementation of Block.run() without using a Gosu block
    * var myBlock = new PluginCallbackHandler.Block() {
    *     override public function run() {
    *         var message = _messageFinder.findById(msgId)
    *         if (message != null) {
    *             message.skip()
    *         }
    *     }
    * }
    */
    _callbackHandler.execute(myBlock)
}
}

```

The following statements show how custom configuration code can call the `skip` plugin method.

```

var messageId = 12345
var myMsgReplyPlugin = gw.plugin.Plugins.get(MyMessageReplyPlugin)
myMsgReplyPlugin.skip(messageId)

```

Error handling in messaging plugins

Several methods on messaging plugins execute in a strict order for a message.

1. ClaimCenter selects a message from the send queue.
2. If this destination defines a `MessageRequest` plugin, ClaimCenter calls `MessageRequest.beforeSend`. This method uses the text payload in `Message.payload` and transforms it and returns the transformed payload. The message transport method uses this transformed message payload later.
3. If `MessageRequest.beforeSend` made changes to the `Message` entity or other entities, ClaimCenter commits those changes to the database, assuming that method threw no exceptions. The following special rules apply.
 - Committing entity changes is important if your integration code must choose among multiple pooled outgoing messaging queues. If errors occur, to avoid duplicates the message must always resend to the same queue each time. If you require this approach, add a data model extension property to the `Message` entity to store the queue name. In `beforeSend` method, choose a queue and set your extension property to the queue name. Then, your main messaging plugin (`MessageTransport`) uses this property to send the message to the correct queue.
 - If exceptions occur, the application rolls back changes to the database and sets the message to retry later. There is no special exception type that sets the message to retry (an un-retryable error).
 - In all cases, the application never explicitly commits the transformed payload to the database.
4. ClaimCenter calls `MessageTransport.send(message, transformedPayload)`. The `Message` entity can be changed, but the transformed payload parameter is read-only and effectively ephemeral. If you throw exceptions from this method, ClaimCenter triggers automatic retries potentially multiple times. This is the only way to get the automatic retries including the backoff multiplier and maximum retries detection.
5. If this destination defines a `MessageRequest` plugin, ClaimCenter calls `MessageRequest.afterSend`. This method can change the `Message` if desired.
6. Any changes from the previous `send` and `afterSend` methods commit if and only if no exception occurred. Any exception rolls back changes to the database and set the message to retry. Be aware there is no special exception class that sets the message not-to-retry.

7. If this destination defines a `MessageReply` plugin, its callback handler code executes separately to handle asynchronous replies. Any changes to the `Message` entity or other entities commit to the database after the code completes, assuming the callback throws no exceptions.

If there are problems with a message, you do not necessarily need to throw an exception in all cases. For example, depending on the business logic of the application, it might be appropriate to skip the message and notify an administrator. If you need to resume a destination later, you can do that using the web services APIs.

All Gosu exceptions or Java exceptions during the methods `send`, `beforeSend`, or `afterSend` methods imply retryable errors. However, the distinction between retryable errors and non-retryable errors still exists if submitting errors later in the message's life cycle. For example, you can mark non-retryable errors while acknowledging messages with web services or in asynchronous replies implemented with the `MessageReply` plugin.

Submitting errors for messages

If there is an error for the message, call the `reportError` method on the message. There is an optional method signature to support automatic retries.

Handling duplicate messages

Your code must handle the possibility of receiving duplicate messages at the plugin layer or at the external system. Typically the receiving system detects duplicate messages by tracking the message ID and returns an appropriate status message. The plugin code that receives the reply messages can report the duplicate with `message.reportDuplicate()` and skip further processing.

Depending on the implementation, your code that receives the reply messages is either within the `MessageTransport.send()` method or in your `MessageReply` plugin.

Saving the destination ID for logging or errors

Each messaging plugin implementation must implement a `setDestinationID` method which receives and stores a unique numeric destination ID in a private variable. The destination can subsequently retrieve and use the destination ID in the following situations.

- Logging code which records the destination ID
- Exception-handling code that works differently for each destination
- Other integrations, such as sending the destination ID to external systems so that they can suspend/resume the destination, if appropriate

Suspend, resume, and shut down a messaging destination

The messaging plugins define methods that are called whenever a destination's active status changes.

Status change	Plugin method called
Suspend destination	<code>override function suspend() : void</code>
Resume destination	<code>override function resume() : void</code>
Shutdown destination	<code>override function shutdown() : void</code>

Typically, an administrator tasked with managing messaging destinations is the person that enacts the suspend and resume operations through the **ClaimCenterAdministration** tab. It is also possible for an external system, using web services provided by ClaimCenter, to perform these operations.

A destination is shut down whenever its server is either physically shut down or its server/system run level is changed. When the server/system run level is changed, the server is symbolically shut down and immediately restarted at the new run level. As part of the shut-down process, the existing messaging plugin instances are not destroyed. When the sending of messages resumes under the new run level, ClaimCenter continues to use the original plugin instances.

IMPORTANT: A plugin's suspend, resume, and shutdown methods must not call the ClaimCenter MessagingToolsAPI web services that suspends or resumes the messaging destination. Guidewire explicitly does not support such circular application logic.

Implementing a simple logging scheme

The following sample code overrides a plugin's suspend method to implement simple logging.

```
override function suspend() : void {
    if(_logger.isDebugEnabled()) {
        _logger.debug("Message transport plugin: Suspending")
    }
}
```

Resuming a messaging destination

If a message transport encounters an exception while resuming its operation (while executing the resume method), the message destination moves to a state that requires manual intervention to resolve. However, it is possible to force the message destination to suspend itself only if the resume method throws an exception. In this case, catch the resume exception and re-throw the exception as the following exception type:

```
gw.plugin.messaging.InitializationException
```

This action forces the message destination into a suspended state. Upon removal of the cause of the exception, the message destination resumes its operation.

Map message payloads by using tokens

About this task

A messaging plugin might need to convert items in the payload of a message, such as typecodes, before sending the message on to the final destination. For many properties governed by typelists, a typecode might have the same meaning in both systems. However, for typecodes that do not match across systems,, you need to map codes from one system to another. For example, convert code A1 in ClaimCenter to the code XYZ for the external system.

If you implement your plugin in Java, you can use a utility class included in the ClaimCenter Java API libraries that map the message payload by using text substitution. The class, `gw.pl.util.StringSubstitution`, scans a message payload to find any strings surrounded by delimiters that you define and then substitutes a new value.

Procedure

1. Choose start and end delimiters for the text to replace.
For example, you can use the 2-character string “**” as the start delimiter and end delimiter. For production code, you might want to use multiple special characters in a sequence that is forbidden in a real field.
2. Put these delimiters around the original text that you need to map and replace.
For example, a Gosu template that generates the payload might include "Injury=**\$ {exposure.InjuryCode}**". This delimited text might generate text such as "Injury=**A1**" in the message payload.
3. Implement a class that implements the inner interface `StringSubstitution.Exchanger`.
This exchanger class must translate exactly one token, not the entire `String` object for the payload. This class might use its own look-up table, a `java.util.Map` object, or look in a properties file. The `Exchanger` interface has one method called `exchange` that translates the token. This method takes a `String` object (the token) and translates it and returns a new `String` object. If the input token requires no substitution, you must decide whether to quietly return the original `String`, or throw an exception.
4. Instantiate your class that implements `Exchanger`, and then instantiate the `StringSubstitution` class with the constructor arguments as follows.

- a. Start delimiter
 - b. End delimiter
 - c. Your Exchanger instance
5. On the new `StringSubstitution` instance, call the `substitute` method to convert the message payload.

Example

The following example demonstrates this process. You can paste the following code into the Gosu Scratchpad in Studio.

```
uses gw.pl.util.StringSubstitution

class TestExchanger implements StringSubstitution.Exchanger {
    public function exchange(s : String) : String {
        print("exchange() method called with token: " + s)
        if (s == "cat")
            { return "kitten"}
        else if (s == "dog")
            { return "puppy"}
        else return s
    }
}
var origString = "wolf cat **cat** dog dog **dog** llama llama **llama**"
var myExchanger = new TestExchanger()
var mySub = new StringSubstitution("/**", "**", myExchanger)
var output = mySub.substitute(origString)

print("final output is: " + output)
```

The example code prints the following output.

```
exchange() method called with token: cat
exchange() method called with token: dog
exchange() method called with token: llama
final output is: wolf cat kitten dog dog puppy llama llama llama
```

Message status code reference

The following table describes the message status values and the contexts in which they can appear for `Message` and `MessageHistory` objects. The first column indicates the static property that you can use on the `gw.pl.messaging.MessageStatus` class to make your code easier to understand.

MessageStatus static property	Message status value	Meaning	Valid in Message	Valid in MessageHi story	Can appear during resync
PENDING_SEND	1	Pending send, the initial state for messages.	•		•
PENDING_ACK	2	Messages are set to this state once the <code>MessageTransport.send</code> method completes but the transport has not acknowledged the message. The status remains in this state until acknowledged, an error is received, or it is retried. These messages are asynchronous and are acknowledged outside the message transport, possibly by a message reply plugin or by using messaging tools.	•		•
ERROR	3	Message has been acknowledged with an error. Due to the type of error involved, it is not possible to retry this message. Guidewire has deprecated the use of this message status.			

MessageStatus static property	Message status value	Meaning	Valid in Message	Valid in MessageHi story	Can appear during resync
RETRYABLE_ERROR	4	Message has been acknowledged with a retryable error.	•		•
ACKED	10	Message has been successfully acknowledged.		•	
ERROR_CLEARED	11	The message was in RETRYABLE_ERROR state but the administrator skipped this message.		•	
ERROR_RETRYED	12	Error retried, and the original message is represented as a MessageHistory object. Another message in the Message table represents the clone of this message.		•	
SKIPPED	13	The administrator skipped this message.	•		

Some static properties on the `MessageStatus` class contain arrays of message status values. You can use them to check values with code that is more easily read.

The class also has static methods that take a status (state) value and return `true` if the status is in a list of relevant values. For example, to test if a message was ever acknowledged (including error values), enter the following Gosu code:

```
gw.pl.messaging.MessageStatus.isAcked(Message.Status)
```

The following table lists additional `MessageStatus` static properties and the static methods.

MessageStatus static property or static method	Description
Properties	
ALL_STATES	An array of all states.
ACKED_STATES	An array of acknowledged states, including error states.
ACTIVE_STATES	An array of all active states.
BLOCKING_STATES	An array of active states for messages blocking sends of subsequent messages.
ERROR_STATES	An array of error states.
INACTIVE_STATES	An array of final message states for messages that no longer require processing.
PENDING_STATES	An array of all pending states.
RETRYABLE_STATES	An array of retryable states, PENDING_ACK or RETRYABLE_ERROR.
Methods	
isActive(state)	Returns <code>true</code> if the state is in the array ACTIVE_STATES.
isInFlight(state)	Returns <code>true</code> if the status indicates a message in flight (PENDING_ACK).
isRetryableError(s tate)	Returns <code>true</code> if the status is RETRYABLE_ERROR.
isRetryable(state)	Returns <code>true</code> if the status is in the array RETRYABLE_STATES.
isPending(state)	Returns <code>true</code> if the status is in the array PENDING_STATES.

MessageStatus	Description
static property or static method	
<code>isPendingSend(stat)</code>	Returns true if the status is PENDING_SEND.
<code>e)</code>	
<code>isAcked(state)</code>	Returns true if the status is in the array ACKED_STATES.
<code>isError(state)</code>	Returns true if the status is in the array ERROR_STATES.

Reporting acknowledgments and errors

Message sending error behaviors

Sometimes something goes wrong while sending a message. Errors can happen at two different times.

- Errors can occur during the send attempt as ClaimCenter calls the message sync transport plugin's `send` method with the message.
- For asynchronous replies, errors can also occur in negative acknowledgments.

The following error conditions can occur during the destination `send` process.

- Exceptions during `send` cause automatic retries.

Sometimes a message transport plugin has a send error that is expected to be temporary. To support this common use case, if the message transport plugin throws an exception from its `send` method, ClaimCenter retries after a delay time, and continues to retry multiple times.

The delay time is an exponential wait time (backoff time) up to a wait limit specified by each destination. For safe-ordered messages, ClaimCenter halts sending messages all messages for that combination of primary object and destination during that retry delay. After the delay reaches the wait limit, the retryable error becomes a non-automatic-retry error.

- Errors during `send` for which you do not want automatic retry.

If the destination has an error that is not expected to be temporary, do not throw an exception. Throwing an exception triggers automatic retry. Instead, call the message's `reportError` method with no arguments.

The destination suspends sending for all messages for that destination until one of the following is true.

- An administrator retries the sending manually, and it succeeds this time.
- An administrator removes the message.
- It is a safe-ordered ClaimCenter message and an administrator resynchronizes the claim.

The following error conditions can occur later in the messaging process:

- A negative acknowledgment (NAK)

A destination might get an error reported from the external system (database error, file system error, or delivery failure), and human intervention might be necessary. For safe-ordered messages, ClaimCenter stops sending messages for this combination of primary object and destination until the error clears through the administration console or automated tools.

- No acknowledgment for a long period

ClaimCenter does not automatically time out and resend the message because of delays. If the transport layer guarantees delivery, delay is acceptable. Considering that resending results in message duplicates, the external system might not be able to properly detect and handle duplicates.

For safe-ordered messages, ClaimCenter does not send more messages for the combination of primary object and destination until it receives an acknowledgment (ACK) or some sort of error (NAK).

Submitting ACKs, errors, and duplicates from messaging plugins

To submit an acknowledgment or a negative acknowledgment from a messaging plugin implementation class, use the following APIs.

To report an acknowledgment

Situation	Method call	Description
Success	<code>message.reportAck()</code>	Submits an ACK for this message, which might permit other messages to be sent.
Errors within the send method of your MessageTransport plugin implementation and the error is presumed temporary	Throw an exception within the <code>send</code> method, which triggers automatic retries potentially multiple times.	<p>Automatically retries the message potentially multiple times, including the backoff timeout and maximum tries. After the maximum retries, the application ceases to automatically retry it and suspends the messaging destination.</p> <p>An administrator can retry the message from the Administration tab. Select the message and click Retry.</p>

To report an error

Situation	Method call	Description
Errors in any messaging plugins and no automatic retry is needed	<code>message.reportError()</code>	The no-argument version of the <code>reportError</code> method reports the error and omits automatic retries. An administrator can retry the message from the Administration tab. Select the message and click Retry .
Errors in any messaging plugins and scheduled retry is needed	<code>message.reportError(date)</code>	Reports the error and schedules a retry at a specific date and time. An administrator can retry the message from the user interface before this date. This is equivalent to using the application user interface in the Administration tab at that specified time, and select the message and click Retry . You can call this method from the <code>MessageTransport</code> or when handling errors in your <code>MessageReply</code> plugin implementation.
Errors in any messaging plugins and scheduled retry is needed	<code>message.reportError(category)</code>	<p>Reports the error and assigns an error category from the <code>ErrorCategory</code> typelist. The Administration tab uses the error category to identify the type of error in the user interface. You can extend the typelist to add your own meaningful values.</p> <p>In the base configuration of ClaimCenter, the typelist is empty.</p> <p>You can call this method from the <code>MessageTransport</code> or when handling errors in your <code>MessageReply</code> plugin implementation.</p>

To report a duplicate

Situation	Method call	Description
Message is a duplicate	<code>msgHist.reportDuplicate()</code>	<p>Reports a duplicate. This method is on the message history object (<code>MessageHistory</code>), not the message object. A message history object is what a message becomes after successful sending. A message history object has the same properties as a message (<code>Message</code>) object but has different methods.</p> <p>If your duplicate detection code runs in the <code>MessageTransport</code> plugin (typical only for synchronous sending), use standard database query builder APIs to find the original message. Query the <code>MessageHistory</code> table.</p>

Situation	Method call	Description
		<p>For asynchronous sending with the <code>MessageReply</code> plugin implementation, the <code>MessageFinder</code> interface has methods that a reply plugin uses to find message history entities. The methods use either the original message ID or the combination of sender reference ID and destination ID.</p> <ul style="list-style-type: none"> • <code>findHistoryByOriginalMessageID(originalMessageId)</code> - find message history entity by original message ID • <code>findHistoryBySenderRefID(senderRefID, destinationID)</code> - find message history entity by sender reference ID <p>After you find the original message in the message history table, report the duplicate message by calling <code>reportDuplicate</code> on the original message.</p>

Using web services to submit ACKs and errors from external systems

If you want to acknowledge a message directly from an external system, use the web service method `MessagingToolsAPI.ackMessage`.

First, create an `Acknowledgement` SOAP object to pass as the `ack` parameter of the method.

There is an `ackCode` that is written to the `MessageHistory` record.

You might have added object names when creating the message in the rules, such as `message.putEntityByName("claim", claim)`. If so, you can change the property values by using `FieldChanges` with that entity name and `FieldChangeValue` entries with the property name and new serialized value. Alternatively, you can raise events on the object by adding `CustomEvents` objects with the entity name and the events to raise.

If there are no problems with the message (it is successful), pass the object as is.

If you detect errors with the message, set the following properties.

- `Error` – Set the `Acknowledgement.Error` property to `true`.
- `Retryable` – For all errors other than duplicates, set the `Acknowledgement.Retryable` property to `true` and `Acknowledgement.Error` to `true`. Set this property to the default value `false` if there is no error.
- `Duplicate` – If you detect that the message is a duplicate, set the `Duplicate` and `Error` properties to `true`.

See also

- “Acknowledging messages” on page 404
- “Saving attributes of a message” on page 379
- “Custom events from SOAP acknowledgments” on page 371

Using web services to retry messages from external systems

The `MessagingToolsAPI` web service contains methods to retry messages.

Review the documentation for the `MessagingToolsAPI` methods `retryMessage` and `retryRetryableErrorMessages`. The method `retryRetryableErrorMessages` optionally limits retry attempts to a specified destination. You can only use the `MessagingToolsAPI` interface if the server’s run mode is set to `multiuser`. Otherwise, all these methods throw an exception.

As part of an acknowledgment, the destination can update properties on related objects. For example, the destination could set the `PublicID` property based on an ID in the external system.

See also

- “Retrying messages” on page 405

Message error handling

ClaimCenter provides several tools for handling errors that occur with sending messages.

- Automatic retries of sending errors
- Ability for the destination to request retrying or skipping messages in error
- User interface screens for viewing and taking action on errors
- Web service APIs for taking action on errors
- A command line tool for taking action on errors

The following kinds of errors can occur. Also described are the actions that can be taken to handle the errors.

Pending Send

The message has not been sent yet.

- If the message is related to a claim, this message is safe-ordered. The messaging destination might be waiting for an acknowledgment on the previous message for that same claim.
- The destination might be suspended, which means that it is not processing messages.
- The destination might not be fast enough to keep up with how quickly the application generates messages. ClaimCenter can generate messages very quickly.

Errors during the send method

ClaimCenter attempted to send the message but the destination threw an exception.

- If the exception was retryable, ClaimCenter automatically attempts to send the message again some number of times before turning it into a failure.
- If the exception indicated a failure, ClaimCenter suspends the destination automatically until an administrator restarts the destination. Failures include a retryable send error reaching its retry limit, or unexpected exceptions during the send method.

The destination also resumes if the administrator removes the message or resynchronizes the message’s primary object.

Pending ACK

ClaimCenter waits for an acknowledgment for a message it sent. If errors occur, such as the external system not receiving or properly acknowledging the message, ClaimCenter waits indefinitely.

If the message has a related primary object for that destination, it is safe-ordered. This type of error blocks sending other messages for that primary object for that destination.

In this case, you can intervene to skip the in-flight message or retry sending it. Be very careful about issuing retry or skip instructions. A retry could cause the destination to receive a message twice. A skip could cause the destination to never get the intended information. In general, you must determine the actual status of the destination to make an informed decision about which correction to make.

To skip or retry sending a message, click the relevant in-flight message link to view the message details screen. Click the **Retry** or **Skip** button.

Error

The destination indicates that the message did not process successfully. The error blocks sending subsequent messages. In some cases, the error message indicates that the error condition might be temporary and the error is retryable. In other cases, the message indicates that the message itself is in error (for example, bad data) and resending does not work. In either case, ClaimCenter does not automatically try to send again.

ACK

The message was successfully processed. The message stays in the system `MessageHistory` table until an administrator purges it.

Note: Since the number of these messages is likely to become very large, Guidewire recommends that you purge completed messages from the `MessageHistory` table on a periodic basis.

Retryable Error

Message sending for the message failed at the external system, not because of a network error. Either the message had an error or was a duplicate.

If ClaimCenter retries a failed message, it marks the original message as failed or retried and creates a copy of the message with a new message ID. A new ID is assigned to the retry message because message destinations can track received messages and ignore duplicate messages.

However, if ClaimCenter retries an in-flight message because it never got an ACK, then it sends the original message again with the same ID. If the destination never got the message, then there is no problem with duplicate message IDs. If the destination received the message but ClaimCenter never got an acknowledgment, then sending the message with the original message ID prevents processing the message twice. The destination can either send back another acknowledgment or refuse to accept the duplicate message and send back an error.

If ClaimCenter receives an error, it holds up subsequent messages until the error clears. If the destination sends back duplicate errors, you can filter out duplicates and warn the administrator about them. However, you can choose to simply issue a positive acknowledgment back to ClaimCenter.

ClaimCenter could become sufficiently out of sync with an external system to make skipping or retrying an individual message insufficient to get both systems in sync. In such cases, you might need special administrative intervention and problem solving. Review your server logs to determine the root cause of the problem.

ClaimCenter makes every attempt to avoid this problem. However, it provides a mechanism called resynchronizing to handle this case. All related pending and failed messages are dropped and resent.

Note: Guidewire ContactManager also supports resynchronization.

Resynchronizing messages for a primary object

ClaimCenter implementations can use the messaging system to synchronize data with an external system.

If a messaging integration condition fails, for example because an external validation requirement was not enforced properly, an external system might process ClaimCenter messages incorrectly or incompletely. If the destination detects the problem, the external system returns an error. The error must be fixed or there can be synchronization errors with the external system.

However, if the administrator fixes the data in ClaimCenter and corrects any related code, the external system might still have incorrect or incomplete data. ClaimCenter provides a programming hook called a *resync* event, resynchronization that recovers from such messaging failures.

To trigger a resync manually, an administrator navigates to the Administration tab in ClaimCenter, views any unsent messages, selects a row, and clicks **Resync**.

The resync can be triggered from the Administration user interface or programmatically by using the `MessagingToolsAPI` web service method `resyncClaim`.

As a result of a resync request, ClaimCenter triggers the event `ClaimResync`. Configure your messaging destination to listen for this event. Then, implement Event Fired business rules that handle that event.

ClaimCenter supports resynchronizing a `Contact` from the user interface. However, in the base configuration, ClaimCenter does not support resynchronizing a contact from web services.

After a resync, ClaimCenter marks all messages that were pending as of the resync as skipped.

You must implement Guidewire Studio rules that examine the data and generate necessary messages. You must bring the external system into sync with the current state of the primary object related to those messages.

Design your resync Event Fired rules to match how your particular external systems recover from such errors.

There are two approaches for generating the resync messages:

- Your Gosu rules traverse all primary object data and generate messages for the entire primary object and its subobjects that might be out-of-sync with the external system.

Depending on how your external system works, it might be sufficient to overwrite the external system's primary object with the ClaimCenter version of this data. In this case, resend the entire series of messages. To help the external system track its synchronization state, it might be necessary to add custom extension properties to various objects with the synchronization state. If you can determine that you need only to resend a subset of messages, send only that minimal amount of information. However, one of the benefits of resync is the opportunity to send all information that might be out of sync. Consider how much data is appropriate to send to the external system during resync.

- Your Event Fired rules that handle the resync can examine the failed messages and all queued and unsent messages for the primary object for a specific destination. Your rules then use that information to determine which messages to re-create. Instead of examining the entire history of the primary object, you can consider only the failed and unsent messages. Because a message with an error prevents sending subsequent messages for that primary object, there can be many unsent pending messages. To help with this process, ClaimCenter includes properties and methods in the rules context on `messageContext` and `Message`.

In your Event Fired rules, your Gosu code can access the `messageContext` object, which contains information to help you copy pending `Message` objects. To get the list of pending messages from a rule that handles the resync event, use the read-only property `messageContext.PendingMessages`. That property returns an array of pending messages. After your code runs, the application skips these original pending messages, and the application permanently removes the messages from the send queue after the resync event rules complete. If there are no pending messages at resync time, this array is empty.

If you create new messages, the new messages are sent in creation order, which might be different from the send order of the original messages. Take into account how this change in order might affect edge cases in the external system.

There are various properties of any message that you can get in pending messages or set in new messages.

payload

A string containing the text-based message body of the message.

user

The user who created the message. If you create the message without cloning the old message, the user by default is the user who triggered the resync. If you create the message by cloning a pending message, the new message inherits the original user who created the original message. In either case, you can choose to set the `user` property to override the default behavior. However, in general Guidewire recommends setting the `user` to the original user. For financial transactions, set the `user` to the user who created the transaction.

There are also read-only properties in pending messages returned from `messageContext.PendingMessages`.

EventName

A string that contains the event that triggered this message. For example, "ClaimAdded".

Status

The message status as an enumeration. Only some values are valid during resync. The utility class `gw.pl.messaging.MessageStatus` provides static properties and static methods that you can use for your code.

ErrorDescription

A string that contains the description of errors, if any. It might not be present. This value is set by a negative acknowledgment (NAK).

SenderRefID

A sender reference ID set by the destination to uniquely identify the message. Your destination can optionally set the `message.senderRefID` field in any of your messaging plugins during original sending of the message. Only the first pending message has this value set due to safe ordering. You need to use the sender reference ID only if it is useful for that external system.

The `SenderRefID` property is read-only from resync rules. This value is `null` unless this message is the first pending message and it was already sent or Pending Send, and it did not yet successfully send. As long as the `message.status` property does not indicate that it is Pending Send, the message could have the sender reference the ID property populated by the destination.

See also

- “Message status code reference” on page 395

Cloning new messages from pending messages

During resync you can clone a new message from a pending message that you received from `messageContext.PendingMessages`. To clone a new a new message from the old message, pass the old message as a parameter to the `createMessage` method.

```
messageContext.createMessage(message)
```

This alternative method signature (in contrast to passing a `String`) is an API to copy a message into a new message and returns the new message. If desired, modify the new message’s properties within your resync rules. All new messages (whether standard or cloned) submit together to the send queue as part of one database transaction after the resync rules complete.

The cloned message is identical to the original message, with the following exceptions.

- The new message has a different message ID.
- The new message has status of pending send (`status = PENDING_SEND`).
- The new message has cleared properties for ACK count and code (`ackCount = 0; ackCode = null`).
- The new message has cleared property for retry count (`retryCount = 0`).
- The new message has cleared property for sender reference ID (`senderRefID = null`).
- The new message has cleared property for error description (`errorDescription = null`).

ClaimCenter marks all pending messages as skipped (no longer queued) after the resync rules complete. Because of this, resync rules must either send new messages that include that information, or manually clone new messages from pending messages, as discussed earlier.

Resync and ClaimCenter financials

Your resync code must handle resending any unprocessed financial transactions, taking care not to resend any successfully processed financial transactions.

How resynchronization affects preupdate and validation

Preupdate and validation rule sets do not run solely because of a triggered event. The claim preupdate and validation rules run only if entity data changes. Triggered events that do not correspond to entity data changes do not cause claim preupdate and validation rules to run.

Most events correspond to entity data changes. However, for events related to resynchronization, no entity data changes. Additionally, custom events that fire through the `addEvent` entity method can also not change entity data.

Resync in ContactManager

If you license Guidewire ContactManager for use with ClaimCenter, be aware that ContactManager supports resync features for the `ABContact` entity.

To detect resynchronization of an `ABContact` entity, set your destination to listen for the `ABContactResync` event.

Your Event Fired rules can detect that event firing and then resend any important messages. Your rules generate messages to external systems for this entity that synchronize ContactManager with the external system.

Monitoring messages

ClaimCenter provides a simple user interface to view the event messaging status for claims. This helps administrators understand what is happening, and might give some insight to integration problems and the source of differences between ClaimCenter and a downstream system.

For example, if you add an exposure to a claim and it does not appear in the external system, you need to know the following information:

- As far as ClaimCenter knows, have all messages been processed? (“Green light”) If the systems are out of sync, then there is a problem in the integration logic, not an error in any specific message.
- Are messages pending, so you simply need to wait for the update to occur. (“Yellow light”)
- Is there an error that needs to be corrected? (“Red light”) If it is a retryable error, you can request a retry. This might make sense if the external system caused the temporary error. For example, perhaps a user in the external system temporarily locked the claim by viewing it on that system’s screen. In many cases, you can simply note the error and report it to an administrator.

This status screen is available from the claim screen by selecting **Claim Actions > Sync Status** from the **Claim** menu.

The ClaimCenter **Administration** tab provides administrators with access to messages sent across the system. Selecting **Monitoring > Message Queues** shows a list of message destinations. Multiple levels of detail are provided for viewing events and messaging status.

Messaging tools web service

ClaimCenter provides the web service **MessagingToolsAPI**, which enables an external system to call the web service methods and remotely control the messaging system.

Note: For administrators, most of the **MessagingToolsAPI** methods are available at the command prompt through the `messaging_tools` command.

See also

- “`messaging_tools` command” in the *Administration Guide*

Acknowledging messages

To acknowledge a message, use the following method:

```
ackMessage(ack : Acknowledgement)
```

The `Acknowledgement` parameter is a SOAP object that passes information about the status of the message.

The method returns `true` if the message is found and acknowledged. If not, the method returns `false`.

The method can throw the following exceptions:

- If the ACK is invalid, the method throws `IllegalArgumentException`.
- If there are permission or authentication issues, the method throws `WsAuthentificationException`.
- If the `Acknowledgement` object is not valid, the method throws `SOAPSenderException`.
- If the ACK could not be committed to the database, the method throws `SOAPException`.

See also

- “Using web services to submit ACKs and errors from external systems” on page 399
- “Messaging tools web service” on page 404

Getting the ID of a message

You can get the ID of a message either from an unspecified destination or from a specific destination. In both cases, you must specify the `senderRefID`.

A sender reference ID is set by the destination to uniquely identify the message. Your destination can optionally set the `message.senderRefID` field in any of your messaging plugins during original sending of the message. Only the first pending message has this value set due to safe ordering.

To get a message ID for a message without specifying the message destination, call the following method with `destID` set to `-1`:

```
get messageId(senderRefID : String, destID : int)
```

If there are multiple messages that match, this method returns the message ID from the first match. If no message is found with the specified `senderRefID` and `messageID`, the method returns `-1`.

To get a message ID for a message sent to a specific message destination, call the following method:

```
get messageIdBySenderRefId(senderRefId: String, destID: int)
```

If there are multiple messages that match, this method throws an exception. If no message is found, the method returns `null`.

These methods can throw the following exceptions:

- If there are permission or authentication issues, the method throws `WsiAuthenticationException`.
- If the destination ID is invalid, the method throws `IllegalArgumentException`.

See also

- “Message status code reference” on page 395
- “Messaging tools web service” on page 404

Retrying messages

You can retry sending messages. There are several variations of retry methods that you can call.

Retrying a single message

To retry sending a message that has a Retryable error state or is in flight, call the following method:

```
retryMessage(messageID : long)
```

The method returns a Boolean value that indicates whether or not the message was successfully submitted for another attempt.

- If the message with this `messageID` does not exist, the method returns `false`.
- Returning `true` means that the message was retried. It does not indicate whether the retry was successful.

If there are permission or authentication issues, the method throws `WsiAuthenticationException`.

Retrying messages for a given destination

To retry all messages that are in the Retryable error state for a given destination, call the following method:

```
retryRetryableErrorMessages(destID : int)
```

The method returns a Boolean value that indicates whether or not the message was successfully submitted for another attempt.

The method throws the following exceptions:

- If there are permission or authentication issues, the method throws `WsiAuthenticationException`.
- If the destination ID is invalid, the method throws `IllegalArgumentException`.

[Retrying messages for a given destination and error category](#)

To retry all messages for a given destination that have a specified error state , call the following method:

```
retryRetryableErrorMessagesForCategory(destID : int, category : ErrorCategory)
```

The method returns a Boolean value that indicates whether or not the message was successfully submitted for another attempt.

The method throws the following exceptions:

- If there are permission or authentication issues, the method throws `WsiAuthenticationException`.
- If the ACK is invalid, the method throws `IllegalArgumentException`.

[Retrying messages for a given destination with Retryable error state and a retry limit](#)

You can retry all messages for a given destination that are in Retryable error state and have been retried fewer times than a specified retry limit. Each message maintains a retry count. Each attempt to retry a message increments its retry count. A message with a retry count that is greater than the retry limit specified in the method call is not retried, unless the retry limit is 0.

Call the following method:

```
retryRetryableSomeErrorMessages(destID : int, retryLimit : int)
```

Note: Specifying a `retryLimit` of 0 retries all retryable error messages and is identical to `retryRetryableErrorMessages(int destID)`.

The method returns a Boolean value that indicates if all messages were successfully submitted for another attempt. If any messages to be retried exceeded the retry limit, the method returns `false`.

The method throws the following exceptions:

- If there are permission or authentication issues, the method throws `WsiAuthenticationException`.
- If the ACK is invalid, the method throws `IllegalArgumentException`.

[See also](#)

- “Messaging tools web service” on page 404
- “Message error handling” on page 400

[Skipping a message](#)

To skip a message, effectively removing an active message from the message queue, call the following method:

```
skipMessage(messageID : long)
```

The method returns a Boolean value indicating whether the message was successfully skipped:

- If the method returns `true`, the message was successfully skipped.
- If the method returns `false`, there can be multiple reasons:
 - The message with this `messageID` does not exist.
 - The message is not in one of the following active states: Pending Send, Inflight, Error, Retryable Error, or Pending Retry.

If there are permission or authentication issues, the method throws `WsiAuthenticationException`.

[See also](#)

- “Messaging tools web service” on page 404
- “Message error handling” on page 400

Resynchronizing a claim for a destination

If ClaimCenter attempts to send a message, but the destination throws an exception, you can retry the message if the exception is retryable. If the message is a failure, ClaimCenter suspends the destination automatically until an administrator restarts it. Failures include a retryable send error reaching its retry limit and unexpected exceptions during the send method.

One way to resume the destination is to resynchronize the message’s primary object. If the primary object is a claim, you can call the following method:

```
resyncClaim(destID : int, claimID : String)
```

IMPORTANT: Your resync code must handle resending any unprocessed financial transactions, taking care not to resend any successfully processed financial transactions.

The method can throw the following exceptions:

- If the claim cannot be found, the method throws `DataConversionException`.
- If there are permission or authentication issues, the method throws `WsiAuthenticationException`.

See also

- “Resynchronizing messages for a primary object” on page 401
- “Messaging tools web service” on page 404

Purging completed messages

You can purge the completed messages stored in the `MessageHistory` table from the ClaimCenter database. Purging completed messages prevents the database from growing unnecessarily large with inactive messages. Additionally, you can reduce the time required to perform a ClaimCenter upgrade by purging completed messages prior to the upgrade.

A completed message has the state `Acked`, `ErrorCleared`, `Skipped`, or `ErrorRetried`.

Call the following method:

```
purgeCompletedMessages(cutoff : Date)
```

The `cutoff` argument is expected to be a date prior to the current date, and it cannot be `null`. A completed message is purged if its send time occurred before the date specified in the `cutoff` parameter.

Note: If the `cutoff` argument is in the future, a date after the current date, the entire `MessageHistory` table is purged and no exception is thrown.

The method can throw the following exceptions:

- If the `cutoff` argument is `null`, the method throws `IllegalArgumentException`.
- If there are permission or authentication issues, the method throws `WsiAuthenticationException`.

See also

- “Messaging tools web service” on page 404
- “Message error handling” on page 400

Suspending a destination

Suspending a destination typically means that ClaimCenter stops sending messages to a destination, although you can stop inbound message processing as well. You can use this method to shut down the destination system and halt

sending during processing of a daily batch file. ClaimCenter suspends the destination so that it can also release any resources such as a local batch file.

When outbound processing is suspended, the request and transport plugins are suspended, along with message sending. When inbound processing is suspended, the reply plugin is suspended.

To see if a destination is suspended, call the following method:

```
isSuspended(destID: int, direction : MessageProcessingDirection)
```

The `direction` parameter can have the `MessageProcessingDirection` values `inbound`, `outbound`, or `both`.

The method returns `true` if processing for the specified destination and direction is suspended. It returns `false` otherwise.

The method can throw the following exceptions:

- If there are permission or authentication issues, the method throws `WsIAuthenticationException`.
- If the ACK is invalid, the method throws `IllegalArgumentException`.

To suspend both inbound and outbound messages for a destination, call the following method:

```
suspendDestinationBothDirections(destID : int)
```

The method can throw the following exceptions:

- If there are permission or authentication issues, the method throws `WsIAuthenticationException`.
- If the destination ID is invalid, the method throws `IllegalArgumentException`.

To choose which direction you want to suspend, call the following method:

```
suspendDestination(destID : int, direction : MessageProcessingDirection)
```

The `direction` parameter can have the `MessageProcessingDirection` values `inbound`, `outbound`, or `both`.

The method returns `true` if processing was previously active and is now suspended. It returns `false` if processing was already suspended.

The method can throw the following exceptions:

- If the `direction` parameter is not a valid value, the method throws `SOAPSenderException`.
- If there are permission or authentication issues, the method throws `WsIAuthenticationException`.
- If ACK is invalid, the method throws `IllegalArgumentException`.
- If processing was in error, the method throws `SOAPException`.

See also

- “Overview of message destinations” on page 346
- “Message error handling” on page 400

Resuming a destination

Resuming a destination generally means that ClaimCenter starts trying to send messages to the destination again. Resuming outbound processing resumes the request and transport plugins and resumes message sending. Resuming inbound processing resumes the reply plugin.

If a previous suspend action released any resources, resuming the destination reclaims those resources. For example, the destination might reconnect to a message queue.

To resume both inbound and outbound messages for a destination, call the following method:

```
resumeDestinationBothDirections(destID : int)
```

The method can throw the following exceptions:

- If there are permission or authentication issues, the method throws `WsiAuthenticationException`.
- If the destination ID is invalid, the method throws `IllegalArgumentException`.

To choose which direction you want to resume, call the following method:

```
resumeDestination(destID: int, direction : MessageProcessingDirection)
```

The `direction` parameter can have the `MessageProcessingDirection` values `inbound`, `outbound`, or `both`.

The method returns `true` if processing was previously suspended and is now resumed. It returns `false` if processing was already active.

The method can throw the following exceptions:

- If the `direction` parameter is not a valid value, the method throws `SOAPSenderException`.
- If there are permission or authentication issues, the method throws `WsiAuthenticationException`.
- If ACK is invalid, the method throws `IllegalArgumentException`.
- If processing was in error, the method throws `SOAPException`.

See also

- “Overview of message destinations” on page 346
- “Message error handling” on page 400

Getting messaging statistics

You can get information about *message statistics*, the number of failed, retryable error, in flight, and unsent (queued) messages, and messages awaiting retry. You can get either all statistics for a message destination or statistics for a safe-ordered message at a destination.

To get all statistics for a message destination, call the following method:

```
getTotalStatistics(destID : int)
```

To get statistics for a safe ordered message, call the following method:

```
getMessageStatisticsForSafeOrderedObject(
    destID : int,
    safeOrderedObjectId : String)
```

The statistics are returned in a `MessageStatisticsData` object.

These methods can throw the following exceptions:

- If there are permission or authentication issues, the method throws `WsiAuthenticationException`.
- If the ACK is invalid, the method throws `IllegalArgumentException`.

See also

- “Message status code reference” on page 395
- “Messaging tools web service” on page 404

Obtaining the status of a destination

You can obtain information about the status of a message destination server by using the following method from the `gw.api.admin.MessagingUtil` class:

```
getDestinationStatus(destID : int)
```

The method returns the destination status as a `String` value.

Destination status types

TypeList `MessageDestinationStatus` defines the possible message status destinations. Possible code values are:

- `Retrying`
- `Shutdown`
- `Started`
- `Suspended`
- `Suspending`
- `Resuming`
- `SuspendedInbound`
- `SuspendedOutbound`
- `Unknown`

Method behavior

The `getDestinationStatus` method returns the current operational status of the specified destination. The method returns a value of `Unknown` if the ownership or status of the destination is in transition. A message destination can be in transition if the destination is currently being migrated between servers, for example.

The returned value can be up to five seconds stale. This means that the method returns a cached value if the method has been called in the last five seconds. The use of caching in this instance attempts to mitigate the effects of repeated status calls overwhelming the destination node.

It is possible to use the following JVM parameter while starting the application server to override the default value of 5000 milliseconds:

```
-Dgw.messaging.destination.status.refresh.intervalInMillis=...
```

This method can throw the following exceptions:

- `WsiAuthenticationException` - If there are permission or authentication issues
- `IllegalArgumentException` - If the ACK is invalid.

See also

- “Message status code reference” on page 395
- “Messaging tools web service” on page 404

Getting configuration information from a destination

To get configuration information from a messaging destination, call the following method:

```
getConfiguration(destID : int)
```

This information is read from files on disk during server startup. However, it can be modified by web services and command-prompt tools.

The `getConfiguration` method takes a single argument that specifies the destination ID.

The method returns an `ExternalDestinationConfig` object, which contains properties matching the parameters for the `configureDestination` method, such as the polling interval and the chunk size.

The method can throw the following exceptions:

- If there are permission or authentication issues, the method throws `WsiAuthenticationException`.
- If the destination ID is invalid, the method throws `IllegalArgumentException`.

See also

- “Changing messaging destination configuration parameters” on page 411

Changing messaging destination configuration parameters

To change messaging destination configuration parameters on a running server, call the following method:

```
configureDestination(  
    destID : int,  
    timeToWaitInSec : int,  
    maxretries : Integer,  
    initialretryinterval : Long,  
    retrybackoffmultiplier : Integer,  
    pollinterval : Integer,  
    numsenderthreads : Integer,  
    chunksize : Integer )
```

Calling this method restarts the destination with the change to the configuration settings. The command waits for the specified time for the destination to shut down. The method has no return value.

The method’s arguments are:

destID

Destination ID of the destination to suspend.

timeToWaitInSec

Number of seconds to wait for the shutdown before forcing it.

maxretries

The number of automatic retries to attempt before suspending the messaging destination.

initialretryinterval

The amount of time in milliseconds after a retryable error to wait before retrying sending a message.

retrybackoffmultiplier

Amount to increase the time between retries, specified as a multiplier of the time previously attempted. For example, if the last retry time attempted was 5 minutes, and backoff is set to 2, ClaimCenter attempts the next retry in 10 minutes.

pollinterval

The required minimum interval between polls from the start of the previous poll to the start of the next poll.

Each messaging destination pulls messages from the database (from the send queue) in batches of messages on the batch server. The application does not query again until **pollinterval** amount of time passes. After the current round of sending, the messaging destination sleeps for the remainder of the poll interval. If the current round of sending takes longer than the poll interval, then the thread does not sleep at all and continues to the next round of querying and sending. If your performance issues primarily relate to many messages for each primary object for each destination, then the polling interval is the most important messaging performance setting.

numsenderthreads

Number of sender threads for multithreaded sends of safe-ordered messages. To send messages associated with a primary object, ClaimCenter can create multiple sender threads for each messaging destination to distribute the workload. These threads actually call the messaging plugins to send the messages.

ClaimCenter ignores this setting for non-safe-ordered messages because ClaimCenter uses one thread for each destination for these types of messages. If your performance issues primarily relate to many messages but few messages per claim for each destination, then this setting is the most important messaging performance setting.

chunksize

The number of messages to read in a chunk.

The method can throw the following exceptions:

- If there are permission or authentication issues, the method throws `WsAuthenticationException`.

- If the destination ID is invalid, the method throws `IllegalArgumentException`.

See also

- “Message processing cycle” on page 351
- “Messaging tools web service” on page 404

Included messaging transports

Email message transport

The base configuration of ClaimCenter provides a message transport that can send standard SMTP emails.

By default, the `emailMessageTransport` plugin registry is registered with a plugin implementation class that has been deprecated. Instead of using the default registered class, register the Gosu plugin implementation class `gw.plugin.email.impl.JaxbEmailMessageTransport`.

See also

- *Gosu Rules Guide*

Register and enable the console message transport

You can register a message transport provided in the base configuration to write debug messages to the console.

About this task

In the base configuration, ClaimCenter provides a console message transport that writes the message text payload to the ClaimCenter console window. You can register this transport in the Plugin Registry editor in Guidewire Studio and enable the destination to debug integration code that creates and sends messages. Using this transport is particularly useful if you are working on financials APIs and want to see if the events look reasonable.

Procedure

1. In Guidewire Studio, navigate in the **Project** window to **configuration > config > Plugins > registry** and open `consoleTransport.gwp`.
2. Replace the registered Gosu class with `gw.plugin.messaging.impl.ConsoleMessageTransport`.
3. Navigate in the **Project** window to **configuration > config > Messaging** and open `messaging-config.xml`.
4. Select the destination with ID 68, `Java.MessageDestination.ConsoleMessageLogger.Name`.
5. Clear the check box for **Disable destination**.
6. Verify the settings for this destination.

In the base configuration, some useful settings for this destination are the following:

Property	Setting
Transport Plugin	<code>consoleTransport</code>
Max Retries	3
Initial Retry Interval	100
Retry Backoff Multiplier	2
Events	<code>\w*</code>

This configuration specifies that ClaimCenter is to send all events to this destination and trigger Event Fired rules accordingly.

7. Restart the ClaimCenter server.

Results

After restarting the server, watch the console window for messages.

part 5

Financials

ClaimCenter integrates with external systems to track financial transactions. ClaimCenter uses integrations to send information to or receive information from the external systems. Examples of such information are details of financials events and tracking and controlling transaction status changes.

ClaimCenter uses both web services and plugins for data transfer.

chapter 23

Financials integration

ClaimCenter integrates with external systems to track financial transactions.

Available integrations include the following:

- Tracking the status of financial transactions
- Tracking payments
- Managing recovery reserve, recovery, and reserve transactions
- Receiving and processing bulk invoices
- Handling deductions

Financial transaction status and status transitions

The status value of a check or transaction represents the current lifecycle state of a check or financial transaction in ClaimCenter. For example, a check status might have the value `issued` or `pendingvoid`.

The status value tracks and controls the flow of a transaction or check through the ClaimCenter check creation and approval process, and the transaction's subsequent submission to an external system. In the case of checks, the status value also affects the actions that are possible after submission to an external system. For each type of financial transaction object (a check, a reserve, and so on), status codes have specific meanings for each kind of financial transaction.

In most cases, the status of a transaction can escalate in only one direction for two specific status values. For example, reserves can transition from `null` to `pendingapproval`, but not from `pendingapproval` to `null`.

To detect status changes of financial transactions, create a messaging destination that listens for entity status change events such as `CheckStatusChanged`, `PaymentStatusChanged`, `RecoveryStatusChanged`, and so on. You can write business rules in the Event Fired rule set to trigger custom actions, such as logging each change or sending notifications to external systems.

The `EntitynameStatusChanged` events trigger after any code creates a financial transaction entity. For example, if you create a new check in ClaimCenter, ClaimCenter triggers a `CheckAdded` and a `CheckStatusChanged` event.

Messages to external systems can represent actions like the following:

- Sending a check to a check printing service
- Voiding a check in an accounting system
- A brief user notification email

- Sending a message to a mainframe that records changes to all financial transactions

Some status transitions always trigger requests and notifications to an external system. If a financial transaction or a transaction change is ready to send to an external system, ClaimCenter changes the financial transaction's status to a status that indicates the pending change. As the transaction's status changes, ClaimCenter generates an event. In the Event Fired rule set you can handle this event by generating new messages to external systems. As the main changes that triggered the issue commit to the database, any new Message objects also commit to the database.

In a separate thread, ClaimCenter sends the message with the messaging plugins.

After the external system acknowledges the message, you must call methods on the financial objects that indicate that the status transition to another status has completed.

Note: There are restrictions with the kinds of changes that can happen in Event Fired rules and messaging plugins. Also be aware that rule-set triggers and concurrent-data exceptions are disabled in some messaging code.

See also

- “Messaging and events” on page 339
- “Restrictions on entity data in messaging rules and messaging plugins” on page 380

Status transitions for financial transactions

The status of a financial transaction can change for the following reasons.

- Message acknowledgments from an external system

Some status transitions happen as a consequence of successful acknowledgment of a special message by an external system. This does not happen automatically. You must call a financials acknowledgment API for the transition to complete.

- User action

User actions trigger some status transitions, such as initiating a void on a check. If you want to customize PCF files to generate different user actions or current actions in different contexts, you must restrict changes to valid status transitions documented in this topic. Discuss any special needs with Guidewire Customer Support for any edge cases. Other status changes happen if a user edits, deletes, approves, or rejects a transaction, all of which can occur only in certain circumstances listed in this topic by transaction type.

- Financials escalation batch processes

Some transitions occur as a consequence of batch processes that move transactions from one status to another, such as from awaiting submission to submitting.

- External systems update check or bulk invoice status with a web service API

Some transitions happen with the `updateCheckStatus` method, which directly affects checks and indirectly affects associated payments. For example, suppose a bank informs the accounts payable system that a check clears. The accounts payable system in turn can use a web service API to update the ClaimCenter check status to the status `cleared`.

Avoid calling locally-hosted SOAP APIs from a plugin or a rule. There are various problems if you call SOAP APIs that modify entities that are currently in use. Additionally, messaging plugins must avoid calling other APIs that might change the message root entity in another bundle.

- Messaging plugin code calling methods on check or bulk invoice entity

Messaging plugin code from the same server can call domain methods on the check or bulk invoice. The following example Gosu code sets the check status to `voided`.

```
(Message.MessageRoot as Check).updateCheckStatus(checkNumber, issueDate, TransactionStatus.VOIDED)
```

The same operation written in Java is shown in the following example.

```
((Check) Message.getMessageRoot()).updateCheckStatus(checkNumber, issueDate, TransactionStatus.VOIDED)
```

The following Gosu code sets the bulk invoice status to voided.

```
(Message.MessageRoot as BulkInvoice).updateBulkInvoiceStatus(  
    checkNumber, issueDate, TransactionStatus.VOIDED)
```

The same operation written in Java is shown in the following example.

```
((Check) Message.getMessageRoot()).updateBulkInvoiceStatus(  
    checkNumber, issueDate, TransactionStatus.VOIDED)
```

In the previous examples, the `updateBulkInvoiceStatus` method updates the status of the placeholder checks. The status of each placeholder check is automatically updated when the status of the bulk invoice changes.

From plugin code, including messaging plugins, do not call SOAP APIs on the same server. To change the status of an item from plugin code, use the previous check or bulk invoice methods. For other types of transactions, such as Recovery, there is no supported API to directly change the status.

You can use these methods only in the following specific contexts. Using the methods in other contexts is unsupported and has unpredictable results.

- Only from messaging plugins.

Never use these methods from rule sets or other Gosu or Java contexts.

- Only after submitting the acknowledgment for a message.

If this message is associated with status transitions that occur only by calling the financials acknowledgment APIs, call those APIs first before attempting other status changes. Make the calls either in your `MessageTransport` plugin after a synchronous send and acknowledgment or in your `MessageReply` plugin after your asynchronous acknowledgment of the message.

IMPORTANT: Changing the `Status` property directly is not supported. Do not change the `Check.Status` property or any other `transaction.Status` property directly in business rules or any Gosu code. Use only the appropriate message acknowledgments or check status update API. Even in those cases, be careful only to make transitions that this documentation identifies as valid status transitions for each transaction type.

Check and transaction status transitions and message acknowledgment

Events and messaging can affect check and transaction status values. For some status transitions, the successful acknowledgment of the message completes some larger process. For example, during a standard check transfer, the status must transition from Pending Transfer to Transferred. In these cases, the transitions do not occur automatically after the message is acknowledged, but only after your messaging plugins call an API.

The steps happen in the following order for status transitions that happen through message acknowledgments.

1. An action occurs that would reflect a status change. For example, suppose a user is transferring a check to another claim.
2. This action changes the status to a pending status. In the check transfer example, the status would be `pendingtransfer`.
3. ClaimCenter generates a `CheckStatusChanged` event.
4. Your business rules catch this event and generate a message.
5. At a later time, in another thread on a server with the `messaging` server role, the messaging destination (your messaging plugins) send the message to an external system. For example, one messaging destination might represent your check printing service or your accounting system.
6. After the external system responds that it processed that message, the messaging plugins that represent the messaging destination submit an acknowledgment (ACK) for that message using the code:
`message.reportAck()`.
7. Immediately after submitting the ACK, your messaging plugins must tell ClaimCenter that financial entity's status must complete the action. For example, for a check transfer the check's status must change from

`pendingtransfer` to `transferred`. Get a reference to the original financials entity instance and call the special method whose name begins with "acknowledge." For a check transfer, the special API to call is `check.acknowledgeTransfer`. For the other APIs for other financials action, refer to the table after this numbered list. Be careful to call the check or transaction submission methods no more than once for each message ACK.

If you fail to call the special acknowledgment method in the financial entity instance, the financial object inappropriately remains in its status without proceeding.

Some financials status transitions happen as part of message acknowledgment. This is not automatic. You must call a special API in your messaging plugins as part of message acknowledgment.

If the current status is not the previous status it expects, these APIs throw an exception. For example, The `check.acknowledgeSubmission` method throws an error if the check is not in `requesting` status. To avoid exceptions, message code such as asynchronous reply plugins can get the status before calling the API. For example, confirm that a check is still in `requesting` status before calling `check.acknowledgeSubmission`.

8. In some cases, as a consequence of calling the financials acknowledge method, a separate but associated financial transaction changes status. For example, if a check changes status, an associated payment may change status too. This status is sometimes identical to the check's status, for example both have the status `voided`. In other cases, they are different statuses, for example the check status is `issued` and the payment status is `submitted`.

The following table defines which acknowledgment methods to use in your messaging plugins.

Action	Call this method at message acknowledgment time in your messaging plugins	Description
Submit a check	<code>check.acknowledgeSubmission</code>	Updates the check's status to <code>requested</code> if it was <code>requesting</code> , or <code>issued</code> if it was <code>notifying</code> . Updates its payments to <code>submitted</code> . Throws an exception if this check is not in <code>requesting</code> or <code>notifying</code> status.
Transfer a check	<code>check.acknowledgeTransfer</code>	Updates the check's status to <code>transferred</code> . Updates its <code>pendingtransfer</code> payments to <code>transferred</code> . For each transferred payment, updates its onset and offset to <code>submitted</code> . Throws an exception if the check is not in <code>pendingtransfer</code> status.
Void a recovery	<code>recovery.acknowledgeVoid</code>	Acknowledges a message that this recovery was <code>voided</code> . Updates its status to <code>voided</code> . Throws an exception if the recovery is not in <code>pendingvoid</code> status.
Recode a payment	<code>payment.acknowledgeRecode</code>	Acknowledges a message that this payment was <code>recoded</code> . Updates its status to <code>recoded</code> . Updates its onset and offset to <code>submitted</code> .
Submit a reserve, recovery, or recovery reserve	<code>transaction.acknowledgeSubmission</code>	Acknowledges a message that this transaction was <code>submitted</code> . Updates its status to <code>submitted</code> . Throws an exception if this transaction is not in <code>submitting</code> status.
Submit a bulk invoice	<code>bulkInvoice.acknowledgeSubmission(message)</code>	Acknowledges a message that this bulk invoice was <code>submitted</code> . Updates its status to <code>requested</code> . For each line item, updates its status to <code>submitted</code> if it was <code>submitting</code> . Throws an exception if this invoice is not in <code>requesting</code> status.

There is no requirement that there be a one-to-one correspondence between the number of messages and the number of affected financial objects. For example, one message can be a submission message for more than one transaction. In that case, during message acknowledgment, you must call the acknowledgment domain method on each relevant financial transaction.

If you add reserves and checks to an exposure that has too low a validation level, you might want to suppress messages to an external system. In most cases, you must not send messages to an external system about an exposure that the mainframe does not know about yet. Later, if the exposure passes a higher validation level, the Event Fired rules must send information about all financial transactions already entered for the exposure. As a result, one message can be associated with multiple events or transactions.

In event fired rules, treat status transition as final

In your Event Fired business rules that catch changes to the status of a check or transaction, treat the current status as final to avoid race conditions. Specifically, assume the status transition is final as soon as the event triggers, even if your messaging code did not yet sent this information to an external system. You must consider the status fully complete and unchangeable within the Event Fired rule sets that generate financials-related messages to external systems.

On a related note, never directly change the `check.status` or any other `transaction.status` property.

However, strictly speaking the change of the financial entity's status does not commit to the database until all related code finishes, including all Event Fired rules and any related validations. If there is a major error in the transaction—for instance, an unhandled exception—the entire transaction rolls back. Thus, any new messages did not commit to the Send Queue, just as the change the check or financial transaction did not change nor commit to the database.

In all cases, the Event Fired rules must consider the transaction final. Any messages that you create commit if and only if the change that triggered the rules commits. This is the correct behavior to ensure ClaimCenter stays in sync with other systems.

If there were no errors and everything commits to the database, any new messages commit to the send queue. The messaging server delivers messages one by one to each destination as separate database transactions.

Debugging financials messaging

ClaimCenter includes a built-in console message transport example, which is an extremely simple messaging transport that writes the message text payload to the ClaimCenter console window. Enable this transport in `config.xml` to debug integration code that creates and sends messages. This is particularly useful if you are working on financials to see if the events look reasonable.

Claim financials web service data transfer objects

ClaimCenter WS-I web services do not support entity data directly as method arguments or return values. Instead, web service APIs use a combination of the following items.

- A String public ID which refers to an object of a specific type by its unique `PublicID` property
- A Gosu class data transfer object (DTO) to encapsulate entity data

For example, instead of passing a `BulkInvoice` entity directly as a method argument, the API might pass a Gosu class called `BulkInvoiceDTO` which acts as the DTO. The DTO class contains only the properties in a `BulkInvoice` entity instance that are necessary for the web service.

If you extend the entity data model with properties that must exist in the web service for sending or receiving data, you must also extend the data transfer object. Add to the set of properties in the corresponding Gosu class with the DTO suffix. For example, if you add an important property to the `BulkInvoice` entity, also add the property to the Gosu class `BulkInvoiceDTO`. Before editing a DTO file, carefully review the introductory comments at the top of the file.

Claim financials web service (`ClaimFinancialsAPI`)

The `ClaimFinancialsAPI` web service manipulates checks, transaction sets, and other financial information attached to claims. To manipulate claims and exposures from external systems in general ways, use the `ClaimAPI` WS-I web services.

There are two general ways to use the `ClaimFinancialsAPI` web service for adding claim financials to a claim.

- Importing financials – Import financials that were already processed in another system. Although validation is run at commit time, there is no programmatic access to validation results other than exception messages.
- Creating financials – Process, approve, and runs validation just like the user interface would do for new transactions. There is full programmatic access to validation results.

These methods work only with active claims. If an external system tries to post updates on an archived claim, ClaimCenter throws an exception.

To check whether a claim is archived, call the `ClaimAPI` web service method `getClaimInfo` to retrieve the archived state.

```
archiveState = myClaimAPI.getClaimInfo("ABC:12345").getArchiveState();
```

If needed, you can use the `reopenClaim` method of the `ClaimAPI` web service to reopen the claim which would enable you to add financials.

See also

- “Claim-related web services” on page 149
- “Getting information from claims/exposures from external systems” on page 151
- “Archiving claims from external systems” on page 154

Importing processed financials from external systems

If financials entry happens outside ClaimCenter, you can import the financials by calling the `ClaimFinancialsAPI` web service method `importClaimFinancials`. Import financials only if financials entry happens outside ClaimCenter.

You can only import transaction sets with status approved. All checks in the transaction set must have a status of requested, pendingvoid, voided, pendingstop, stopped, issued, or cleared. All other types of transactions in the transaction set must have a status of submitted, pendingvoid, voided, pendingstop, stopped, or recoded.

Never attempt to import financials that would fail validation. The method does not perform the additional steps of duplicate checking, approval processing, or submission procession. Validation rules are performed when committing to the database, but are not performed when importing financials from an external system.

For checks, if a claim or exposure is not at the Ability to Pay validation level, a built-in approval rule prevents check approval and sets check status to Pending Approval.

The transaction set can attach a set of `Document` objects to the claim.

- If the documents are already in the database, populate the document public ID String values in the `TransactionSetDTO.DocumentIDs` property.
- If the documents are not already in the database, populate a set of `DocumentDTO` objects and include them in the `TransactionSetDTO.NewDocuments` property.

Creating new financials from external systems

To create new financial transactions in an automated way just as they would from the user interface, use the `ClaimFinancialsAPI` web service methods that start with the prefix `create`. For example, `createCheckSet`. These methods run approval and Transaction Post-Setup rules and provide full programmatic access to validation results.

Some additional requirements exist for the financials creation methods.

- The items must correspond to a single claim.
- All of the transactions must have the same currency in the same request. If the reserving currency is not provided on the item, ClaimCenter sets it to the claim currency.

The behavior of these methods is described below.

- Creation methods that create financials run approval and Transaction Post-Setup rules.

- Creation methods run Validation rules twice: Once before submitting for approval, and once during database commit.
- All the methods encapsulate the main data in a `TransactionSetDTO` data transfer object. The object's `Subtype` property defines what type of transaction set it is.

The following table summarizes the methods.

To create a transaction set of this type	Use this <code>ClaimFinancialsAPI</code> web service method	Description
Check set	<code>createCheckSet</code>	Creates a check set from an external system
Recovery set	<code>createRecoverySet</code>	Creates a recovery set from an external system
Recovery reserve set	<code>createRecoveryReserveSet</code>	Creates a recovery reserve set from an external system
Reserve set	<code>createReserveSet</code>	Creates a reserve set from an external system

All these methods return a validation result object of type `TransactionSetApprovalResult`. That result object has a `State` property that specifies the result state. The rest of the details in the `TransactionSetApprovalResult` depend on the value of the `State` property enumeration.

- `INVALID` – The result contains a list of validation errors. Check the `TransactionSetApprovalResult.ValidationErrors` property for details.
- `VALID_UNAPPROVED` – The result contains the public ID of the imported transaction set as well as a list of reasons for approval. Check the `TransactionSetApprovalResult.ApprovalReasons` property for details.
- `VALID_APPROVED` – The result contains the public ID of the imported transaction set.

If you want approval to run, set the financial transaction status for the financials to the draft status (`TransactionStatus.TC_draft`). For checks and transactions, in general set the financial transaction status for incoming financials to the draft status (typecode `TransactionStatus.TC_draft`).

You can also choose to use `AwaitingSubmission` or `PendingApproval` status to mark the object for Rules to transition a recently-imported check to a later status such as `issued`. Do not set status to a status value that implies committed actions, such as the status `submitted`, `requesting`, `requested`, or `issued`. Those values will throw exceptions on import.

For checks, if a claim or exposure is not at the Ability to Pay validation level, a built-in approval rule prevents check approval and sets check status to Pending Approval.

The transaction set can attach a set of `Document` objects to the claim.

- If the documents are already in the database, populate the document public ID `String` values in the `TransactionSetDTO.DocumentIDs` property.
- If the documents are not already in the database, populate a set of `DocumentDTO` objects and include them in the `TransactionSetDTO.NewDocuments` property.

Multicurrency with new financials

The web service methods that import and create financials support the setting of custom exchange rates. In the `TransactionSetDTO` data transfer object, set the `NewExchangeRate` and `NewExchangeRateDescription` properties before passing the data transfer object to ClaimCenter. All the transactions in the transaction set must have the same exchange rate. The `TransactionDTO` object cannot override a transaction with its own exchange rate.

There is internal validation that the passed-in amount is the same as the calculated value of the product of transaction amount and exchange rate. If they are not equal, ClaimCenter logs a warning message, but still saves the change.

Multicurrency foreign exchange adjustment SOAP APIs

There are two `ClaimFinancialsAPI` web service methods to perform foreign exchange adjustments using web services. There are two different methods, one to apply to a check, and one to apply to a payment. The methods support single-payee and joint-payee checks, but multi-payee checks are not supported.

For a check, use the method `applyForeignExchangeAdjustmentToCheck`. For a payment, use the method `applyForeignExchangeAdjustmentToPayment`.

These methods take the following arguments.

- A check or payment public ID.
- A new claim amount of type `BigDecimal`. If `null`, the claim amount is not adjusted.
- A new reporting amount of type `BigDecimal`. This value is the reporting amount of the check. If `null`, the reporting amount is not adjusted.

When the final converted amount is known, both methods apply a foreign exchange adjustment to the payments on the indicated check or payment. This method can only be called on a check or payment that has already been escalated and sent downstream, but has not been canceled or transferred.

The method is only for the case of where the insurer's deployment does not support multiple base currencies. That is, use the method if the claim and reporting currencies are always the same. For example, suppose you create a check with two payments in the amounts of \$60.00 and \$40.00 in the transaction currency. Initially, the payments have the same amounts in the claim currency, which means the exchange rate at the time of check creation is 1:1. If the check eventually cashes, suppose the exchange rate changes to be 1:1.1. The two payments now total \$110.00 in the claim currency. If this happens, you can call this method and pass \$110 as the new claim currency amount. The additional \$10 divides up between the payments proportionally, resulting in one payment for \$66.00, and one for \$44.00, which is a 10% increase of each. If either payment had multiple line items, the additional monies divide up proportionally across line items. The Total Incurred and Total Payments calculated values change to reflect the foreign exchange rate adjustment.

Check integration

There are two types of checks in ClaimCenter.

Standard checks

The normal checks requested from ClaimCenter when using the **Check Wizard** in the user interface. You can also trigger them from business rules. Typically, you generate checks in the user interface, triggering an approval process of automated approval rules and human approvers. Eventually, ClaimCenter sends checks to external systems to print and process. You can track check status, and request a void or stop if necessary.

Manual checks

Checks that record physical checks written from a check book or checks written from a system external to the ClaimCenter application. In this case, ClaimCenter tracks the check for completeness only. Processing manual checks in ClaimCenter enables you to track checks in a central location with optional notification of other users or external systems. You can track check status, and request a void or stop if necessary. Manual checks can be assigned a Notifying status that indicates that printing the check is unnecessary.

Technically, checks are not transactions. However, in most ways ClaimCenter treats checks like other financial transactions entities, such as tracking them with a status code.

The following table describes check status codes. The columns on the right indicate whether the status exists for standard checks or manual checks.

Check status	Meaning	Standard	Manual
<code>null</code>	An internal initial status.	Yes	Yes

Check status	Meaning	Standard	Manual
pendingapproval	The check request is saved in ClaimCenter but is not yet approved. Manual checks do not need approval. However, you can customize this behavior if necessary.	Yes	Yes
rejected	The approver did not approve the check request. In the base configuration, manual checks do not need approval, but you can customize this behavior.	Yes	Yes
awaitingsubmission	The check is held in this status until the date reaches the check's issuance date and a background processing task prepares the check for submission.	Yes	--
requesting	The standard check request is ready to be sent to an external system.	Yes	--
requested	The standard check was successfully sent to an external system.	Yes	--
pendingvoid	The check void request is ready to be sent to an external system.	Yes	Yes
pendingstop	A check stop request is ready to be sent to an external system.	Yes	Yes
pendingtransfer	A check transfer request is ready to be sent to an external system.	Yes	Yes
issued	The check was issued in the external system. For manual checks, <i>issued</i> means specifically that the manual check notification was sent to an external system and successfully acknowledged.	Yes	Yes
cleared	The check cleared in the external system.	Yes	Yes
notifying	The manual check notification is ready to be sent to an external system, indicating that printing the check is unnecessary.	--	Yes

To detect standard check requests or manual checks, you can write event business rules that listen for the `CheckStatusChanged` event and check for changes to the `check.status` property. The following table describes the possible status code transitions. The last two columns indicate whether the transition exists for standard checks or manual checks.

Original check status	Changed status	Status change description	Standard	Manual
null	pendingapproval	For a new check, either approval rules triggered approval or the check exceeded authority limits. In the base configuration, ClaimCenter auto-approves manual checks. You can customize this behavior with approval rules.	Yes	Yes
	awaitingsubmission	A standard check reaches the end of the new check wizard and approval rules determine that no approval is necessary.	Yes	--
	notifying	A manual check makes this initial transition if the check does not require approval. In the base configuration, ClaimCenter auto-approves manual checks. You can customize this behavior with approval rules.	--	Yes
pendingapproval	awaitingsubmission	The approver approves the standard check.	Yes	--
	rejected	The approver rejects the check.	Yes	Yes
	object deleted	A user deletes a check in the user interface.	Yes	Yes
	notifying	The approver approves the manual check.	--	Yes
rejected	awaitingsubmission	A user's edit to a check does not trigger approval.	Yes	--
	pendingapproval	A user's edit to a check triggers approval. In the base configuration, ClaimCenter auto-approves manual checks. You can customize this behavior with approval rules.	Yes	Yes
	notifying	A user edits a rejected check and the updated check does not require approval.	--	Yes

Original check status	Changed status	Status change description	Standard	Manual
	<i>object deleted</i>	A user deletes a check in the user interface.	Yes	Yes
awaitingsubmission	requesting	Automatically transitions to the new status by using the scheduled financials escalation batch process.	Yes	--
	<i>pendingapproval</i>	A user's edit to a check triggers approval.	Yes	--
	<i>object deleted</i>	A user deletes a check in the user interface.	Yes	--
requesting	requested	For a standard check only, this transition indicates that ClaimCenter received an acknowledgment for the associated message for the requesting status. This transition requires you to call <code>check.acknowledgeSubmission</code> in your message ACK code in your messaging plugins.	Yes	--
	<i>pendingtransfer</i>	A user attempted to transfer a check.	Yes	--
	<i>pendingvoid</i>	A user attempted to void a check in the application user interface.	Yes	--
	<i>pendingstop</i>	A user attempted to stop a check in the application user interface.	Yes	--
	<i>denied</i>	For single standard checks, this transaction works from the check method <code>denyCheck</code> (either SOAP API method or domain method). For single manual checks, this transition works only from the check method <code>denyCheck</code> called from messaging plugins. It does not work from the SOAP API version of this method. This transaction works only for single checks. The check cannot be part of a multipayee/grouped check, nor can it be a <code>BulkInvoiceItem</code> check.	Yes	--
	<i>issued</i>	This transition is not allowed. The check must first have the status requested.	--	--
	<i>cleared</i>	This transition is not allowed. The check must first have the status requested.	--	--
<i>pendingvoid</i>	<i>voided</i>	From a SOAP API or a domain method, <code>updateCheckStatus</code> was called for a check status change that indicates that the check voided.	Yes	Yes
	<i>issued</i>	From a SOAP API or a domain method, <code>updateCheckStatus</code> was called for a check status change that indicates that the check was issued. The void was unsuccessful.	Yes	Yes
	<i>cleared</i>	From a SOAP API or a domain method, <code>updateCheckStatus</code> was called for a check status change that indicates that the check cleared. The void was unsuccessful.	Yes	Yes
	<i>stopped</i>	This transition is not allowed. To simulate this transition, perform the following operations. 1. Transition from PendingVoid to Issued by using the method <code>updateCheckStatus</code> , either the SOAP API method or the identically named domain method.	Yes	Yes

Original check status	Changed status	Status change description	Standard	Manual
		<p>2. Request stop by using the <code>stopCheck</code> method (SOAP API or domain method).</p> <p>3. Use <code>updateCheckStatus</code> again to transition from <code>PendingStop</code> to <code>Stopped</code>.</p>		
<code>pendingstop</code>	<code>stopped</code>	From a SOAP API or a domain method, <code>updateCheckStatus</code> was called for a check status change that indicates that the check stopped.	Yes	Yes
	<code>issued</code>	From a SOAP API or a domain method, <code>updateCheckStatus</code> was called for a check status change that indicates that the check issued. The stop was unsuccessful.	Yes	Yes
	<code>cleared</code>	From a SOAP API or a domain method, <code>updateCheckStatus</code> was called for a check status change that indicates that the check cleared. The stop was unsuccessful.	Yes	Yes
	<code>voided</code>	<p>This transition is not allowed.</p> <p>To simulate this transition, perform the following operations.</p> <ol style="list-style-type: none"> 1. Transition from <code>PendingStop</code> to <code>Cleared</code> by using the method <code>updateCheckStatus</code>, either the SOAP API method or the identically named domain method. 2. Request void by using the <code>voidCheck</code> method (SOAP API or domain method). 3. Use <code>updateCheckStatus</code> again to transition from <code>PendingVoid</code> to <code>Voided</code>. 	Yes	Yes
<code>pendingtransfer</code>	<code>transferred</code>	ClaimCenter received an acknowledgment for the associated message for the <code>pendingtransfer</code> status. This transition requires you to call <code>check.acknowledgeTransfer()</code> in your message ACK code in your messaging plugins.	Yes	Yes
<code>issued</code>	<code>pendingtransfer</code>	A user requested a check transfer.	Yes	Yes
	<code>pendingvoid</code>	A user attempted to void a check in the application user interface.	Yes	Yes
	<code>pendingstop</code>	A user attempted to stop a check in the application user interface.	Yes	Yes
	<code>cleared</code>	SOAP API for a check status change that indicates the check cleared.	Yes	Yes
<code>cleared</code>	<code>pendingtransfer</code>	A user attempted to transfer a check.	Yes	Yes
	<code>pendingvoid</code>	A user attempted to void a check in the application user interface.	Yes	Yes
<code>requested</code>	<code>pendingtransfer</code>	A user attempted to transfer a check.	Yes	--
	<code>issued</code>	From a SOAP API or a domain method, <code>updateCheckStatus</code> was called for a check status change that indicates that the check issued.	Yes	--

Original check status	Changed status	Status change description	Standard	Manual
	cleared	From a SOAP API or a domain method, updateCheckStatus was called for a check status change that indicates that the check cleared.	Yes	--
	pendingvoid	A user requests a check void.	Yes	--
	pendingstop	A user requests a check stop.	Yes	--
	denied	For single standard checks, this transaction works from the check method denyCheck (either SOAP API method or domain method). This transition works only for single standard checks. The check cannot be part of a multipayee/grouped check, nor can it be a BulkInvoiceItem check.	Yes	--
notifying	pendingtransfer	A user requested a check transfer.	--	Yes
	issued	For a manual check only, this transition indicates that ClaimCenter received an acknowledgment for the associated message for the requesting status. This transition requires you to call check. acknowledgeSubmission in message acknowledgment code in your messaging plugins.	--	Yes
	cleared	The check must first have the status issued. For manual checks, this transition happens if, from the SOAP API or domain method, calling updateCheckStatus for a check status change indicates that the check cleared. For standard checks, this transition is not allowed.	--	Yes
	pendingvoid	A user requested a check void.	--	Yes
	pendingstop	A user requested a check stop.	--	Yes
	denied	For single standard checks, this transition happens through the check method denyCheck, either through SOAP APIs or the domain method. For single manual checks, this transition works only from the check method denyCheck called from messaging plugins. It does not work from the SOAP API version of this method. This transition works only for single checks. The check cannot be part of a multipayee/grouped check, nor can it be a BulkInvoiceItem check.	--	Yes
voided	issued	From a SOAP API or a domain method, updateCheckStatus was called for a check status change that indicates that the check issued.	Yes	Yes
	cleared	From a SOAP API or a domain method, updateCheckStatus was called for a check status change that indicates that the check cleared.	Yes	Yes
stopped	issued	From a SOAP API or a domain method, updateCheckStatus was called for a check status change that indicates that the check issued.	Yes	Yes
	cleared	From a SOAP API or a domain method, updateCheckStatus was called for a check status change that indicates that the check cleared.	Yes	Yes

Required message acknowledgments for checks

Message acknowledgments trigger certain automatic status transitions. The messaging plugin representing the receiving system must call special financials acknowledge methods for the transition to occur. For checks, acknowledgment-driven status transitions apply in the following cases:

- Standard check status: `requesting` → `requested`
- Manual check status: `notifying` → `issued`
- Standard check status: `pendingtransfer` → `transferred`

Updating check status

You can update check status from an external system or from a plugin.

Updating check status from an external system

To update a check status from an external system, call the `ClaimFinancialsAPI` web service `updateCheckStatus` method. The method accepts the following arguments.

- Check public ID
- Check number (ignored if null)
- Issue date (ignored if null)
- Status of the check specified as a `TransactionStatus` typecode

An example code statement is shown below.

```
claimfinancialsAPI.updateCheckStatus("abc:1234", "1024", null, TransactionStatus.TC_REQUESTED);
```

Updating check status from a plugin

To update a check status from a plugin, call the `updateCheckStatus` method of the `Check` class. The method accepts the following arguments and has no return value:

String check number

Used when the updated check status is `TC_ISSUED`. Otherwise can be `null`.

java.util.Calendar issue date

Used when the updated check status is `TC_ISSUED`. Otherwise can be `null`.

Updated check status specified as a `TransactionStatus` typecode

Valid `TransactionStatus` settings are `TC_ISSUED`, `TC_CLEARED`, `TC_VOIDED`, and `TC_STOPPED`. If the check is part of a multi-payee check group that has been voided or stopped, then only the `TC_VOIDED` and `TC_STOPPED` settings are valid.

If the updated check status is `TC_VOIDED` or `TC_STOPPED`, then the status of associated `Payment` objects are also updated to the respective `Payment` codes `TC_PENDINGVOID` or `TC_PENDINGSTOP`.

If the updated check status is `TC_ISSUED` or `TC_CLEARED` and the original status is `TC_PENDINGVOID` or `TC_PENDINGSTOP`, then the statuses of payments are also accordingly updated. In addition, a warning activity is created and assigned to the user who attempted to void or stop the check. The warning activity notifies the user that their original action did not occur. Finally, ClaimCenter generates any required reserve necessary to keep open reserves from becoming negative.

If the updated check is linked to a `BulkInvoiceItem` (`check.Bulked` is set to `true`), then updating the check status also updates the `BulkInvoiceItem` status, if appropriate.

Message acknowledgments trigger certain automatic status transitions. The messaging plugin representing the receiving system must call financial acknowledgment methods for the transition to occur. For those transitions, call the acknowledgment method on any related financial objects before attempting any other status transitions.

Handling negative or zero value check amounts

Though it is a rare use case, in the base application, it is possible to create a check with a negative amount or a zero amount in ClaimCenter. More specifically, the amount is the value of `Check.GrossAmount`.

Ensure your integration code is aware of and appropriately handles negative or zero value checks. Generally, this means not sending the check to a check printing system or to an EFT delivery service. However, sending to the general ledger system is acceptable, to ensure Total Paid is reflected correctly.

To update check status for negative or zero value checks, call `check.acknowledgeSubmission` in message acknowledgment code in your messaging plugins.

See also

- *ClaimCenter Application Guide*

Voiding, stopping, denying, and reissuing checks

ClaimCenter includes APIs to change status of a check in specific ways. They are available in two forms.

- The `ClaimFinancialsAPI` web service which can be called by an external system.
- Domain methods on `Check` entity instances.

Voiding a check

To void a check, call the method `voidCheck`. This is available as a method in the `ClaimFinancialsAPI` web service and also as a domain method on a `Check` entity instance. Both fundamentally have the same behavior, although the arguments are slightly different due to how web services work.

The web service variant takes a check public ID and a reason, which you can set to `null` if unneeded.

```
claimfinancialsapi.voidCheck("abc:12345", "this is the reason");
```

The domain version takes no arguments.

```
check.voidCheck();
```

Both versions void a check. The operation changes the status of the check and creates offsetting payments to offset each payment on the check. In addition, the API creates offsetting reserves if a payment on the check is eroding and either of the following conditions is true.

- The payment's exposure is closed. Closed exposures always have zero open reserves.
- Open reserves on the payment's `ReserveLine` would become negative without an offsetting reserve.

Offsetting reserves are included in this check's `CheckSet`.

The status of the check and the original payments on the check are set to `pendingvoid`. The offsetting payments are set to `submitting` status. The status of any offsetting reserves is set to `submitting`.

This method works for both single-payee and multi-payee checks. However, if this is a multi-payee check, then the void request happens for all checks in the check group.

This action does not require approval.

The method throws an exception if the check status is not one of the following: `notifying`, `requested`, `requesting`, `issued`, `cleared`.

Voiding and reissuing a multi-payee check

The method called `voidAndReissueCheck` applies only to multi-payee checks. This method does not affect the payments on the check and does not create any new transaction entities. Contrast this behavior with the method called `voidCheck`, which creates offsetting payments. This operation is available as a method in the `ClaimFinancialsAPI`

web service and also as a domain method on a Check entity instance. Both have the same behavior, but the arguments are slightly different due to how web services work.

The method voids and reissues the check. ClaimCenter creates a new replacement check. The status of the original check becomes pendingvoid.

The voidAndReissueCheck method takes an extra parameter for the description of the reason for stopping the check. For example, the following code calls the SOAP API version of voidAndReissueCheck.

```
claimfinancialsapi.voidAndReissueCheck("abc:12345",
    "Check was accidentally sent twice. Voiding this one.");
```

The action does not require approval.

The API throws an exception if the check status is not one of the following: notifying, requested, requesting, issued, cleared.

Reissuance performs the following operations.

1. If the original check was the primary Check for the CheckGroup, the new check becomes the primary Check. The original check converts to a secondary Check (still in the same CheckGroup). All of the Payments and Deductions move to the new Check.
2. Regardless of whether the original Check already had a CheckPortion, ClaimCenter creates a new, fixed-amount CheckPortion for it. In case it does not already specify a fixed portion, its amount does not fluctuate (for example, if it previously used a percentage portion).
3. The following properties are set on the new Check.
 - CheckNumber and IssueDate are null.
 - ScheduledSendDate is set to the current day.
 - The status is awaitingsubmission.

You can configure how ClaimCenter initializes the new check.

Stopping a check

Stopping checks

The stopCheck method is available as a ClaimFinancialsAPI web service and also as a domain method on a Check entity instance. However, the arguments are slightly different due to how web services work.

The voidCheck and stopCheck methods have similar behavior, except that the stopped check transitions into pendingstop status instead of pendingvoid status. The check status requirements are the same as for voidCheck.

The SOAP API version takes a check public ID and a reason String. The reason argument can be set to null.

```
claimFinancialsApi.stopCheck("abc:12345", "this is the reason");
```

The domain method version takes no arguments.

```
check.stopCheck()
```

Cancelling instant payouts

Stopping a check with the Instant payment method is known as a cancellation, since the aim is to cancel the corresponding payout in the external payment gateway system. This is achieved using the cancelInstantPayment plugin method in the InstantCheckTransportPlugin.gs (Destination ID: 71) plugin class in the base application. When an instant check is stopped in the ClaimCenter user interface, event fired rules detect a change to the check, which call this plugin method. If all is successful, the payout status is changed to CANCELED.

Depending on the payment gateway configuration and what the gateway supports, cancellations can be limited to only certain payout statuses, or not supported at all. Use the supportsStopInstantPayment method in the

`OutboundInstantPaymentGatewayPlugin` plugin to configure this behavior. This method checks the status of the payout and returns true if cancellation is supported. If false, the **Stop Check** button on the check in ClaimCenter is not available.

A call to `cancelInstantPayment` is an attempt to cancel the payout. Depending on the payment gateway, and the status of the payout at the time the cancellation is attempted, there is a set of possible outcomes that may or may not result in a successful cancellation, as shown in the following:

Payout status response	Result in ClaimCenter
CANCELED	Cancellation successful. Check moved from Pending Stop to Stopped status in ClaimCenter.
COMPLETED	Cancellation unsuccessful. Payout had completed previously and funds successfully disbursed to payee.
FAILED	Payout had failed previously in payment gateway. Check moved from Pending Stop to Stopped status in ClaimCenter.
null	Cancellation still being processed or was unsuccessful. Check remains in Pending Stop status in ClaimCenter. An asynchronous call is expected to update the payout status via the API.

Full details of all cancellation operations and method calls, refer to the `InstantCheckTransportPlugin.gs` class found in `gsrsrc > gw > plugin > financials > payment`.

See also

- “Mapping payout status to check status” on page 434

Stopping and reissuing a multi-payee check

Similar to the method called `voidAndReissueCheck`, the method called `stopandReissueCheck` applies only to multi-payee checks. The `stopandReissueCheck` method does not affect the payments on the check and does not create any new transaction entities. This method is the same as `voidAndReissueCheck`, except the stopped check transitions into `pendingstop` status instead of `pendingvoid` status.

The `stopandReissueCheck` method takes an extra parameter for the description of the reason for stopping the check.

```
claimfinancialsapi.stopAndReissueCheck("abc:12345", "fraud was detected after check was created");
```

Denying a check

Denial of a check supports automated processes in downstream systems that catch an invalid check and then deny it immediately. Only single-payee checks can be denied.

An example for denial is catching an invalid payee because the payee is on a watch list. To deny a check, the check must be in the status `requesting` or `requested`. After the check transitions to the status `issued`, `cleared` or further statuses such as `voided`, it is too late to deny the check. Denying the check must happen almost immediately, if needed.

To deny a check from web services, call the `ClaimFinancialsAPI` method `denyCheck`. It takes a check public ID.

To deny a check from Gosu, such as from within a messaging plugin, call the method directly on the `Check` object with no arguments.

```
check.denyCheck()
```

Modifying the check scheduled send date

The check property for the schedule send date, `ScheduledSendDate`, is only modifiable in Transaction Presetup Rules or from Gosu called from the application’s user interface PCF code. The date determines whether to include the check amount in one of the following categories.

- Future Payments – tomorrow or later
- Awaiting Submission – today, which is included in Total Payments and similar financials calculations

If the date is inappropriately updated, ClaimCenter throws an exception with following message.

Check contains a payment whose LifeCycleState is inconsistent with the Check's Scheduled Send Date.

Instant check integration

Payment gateway integration for instant payouts

The instant disbursements feature in ClaimCenter requires integrating with an external payment gateway system to issue instant payouts to individual payees. For example, a disbursement to a prepaid debit card, Zelle, PayPal, and other payment mechanisms supported by the payment gateway system.

To enable this integration (and the **Instant** payment method option in the check wizard), ensure the application configuration parameter `InstantPaymentIntegrationEnabled` is set to `true` in `config.xml`. You must also configure the `OutboundInstantPaymentGatewayPlugin` with an implementation that talks to your external payment gateway system. This plugin initiates a call to the payment gateway to request an instant payout and sends the necessary check information from ClaimCenter to the gateway. Then, the payment gateway issues the funds to the payee. This is known as a payout.

Note: Refer to the `OutboundInstantPaymentGatewayPlugin.java` class found in `src > gw > plugin > financials > paymentgateway` for a complete list of plugin methods available.

Payout trigger

The process to initiate an outbound payout is as follows:

1. A user in ClaimCenter creates a check in the ClaimCenter UI and specifies the **Instant** payment method in the New Check Wizard.
2. A ClaimCenter preupdate rule escalates the instant check status to Requesting.
3. ClaimCenter event messaging rules call the `shouldMakeOutboundPayment()` method in the `InstantCheckUtil.gs` class to determine if a Message entity must be created to send the Check to the payment gateway.
4. In the `InstantCheckTransportPlugin.gs` (Destination ID: 71) plugin class, the Message entity is processed, and an API call to the payment gateway provider initiates the payout.
5. The plugin method returns an `InstantPaymentReference` where the Status field is set to the `InstantPaymentStatus` of INITIATED.
6. The value of Status on the Check entity is updated based on the `InstantPaymentStatus` and other values in `InstantPaymentReference` are stored on the Check.

Note: Once the payout is processed, information on the result of the payout is sent back to ClaimCenter.

For more information on this process, and the mapping of a payout status to a check status, see “Mapping payout status to check status” on page 434.

`initiateInstantPayment()` method

This method in the `OutboundInstantPaymentGatewayPlugin` is an API call to the payment gateway to initiate an outbound instant payout. Its arguments include data transfer objects (DTOs) from the following:

- recipient – defined in `Recipient.java`
- payout – defined in `Payout.java`
- claimNumber (`check.Claim.ClaimNumber`)
- policyNumber (`check.Claim.Policy.PolicyNumber`)
- Map<String, String> metadata – extra information associated with the payout

If necessary, you can configure ClaimCenter to generate an `InstantPmtExternalID`. It is recommended that this value originate from the payment gateway. If you do decide to generate this from ClaimCenter, provide it in the String Map metadata field associated with this method (`@param metadata`).

See also

- *ClaimCenter Application Guide*

Mapping payout status to check status

When a check with the **Instant** payment method is created in ClaimCenter, its lifecycle depends on how its payout counterpart is processed in an external payment gateway system.

After the initial API call to the payment gateway, a payout is processed and transitions through a set of statuses. These statuses are mapped back to the check statuses in ClaimCenter.

Changes to checks in ClaimCenter are communicated to the payment gateway by the `OutboundInstantPaymentGatewayPlugin` plugin.

The list of available payout statuses in the base application is in the `InstantPaymentStatus.java` Java enumeration. In Guidewire Studio, this class is found in `src > gw > api > financials > paymentgateway`. The payout status is returned using an `InstantPaymentReference.java` interface (also found in `src > gw > api > financials > paymentgateway`).

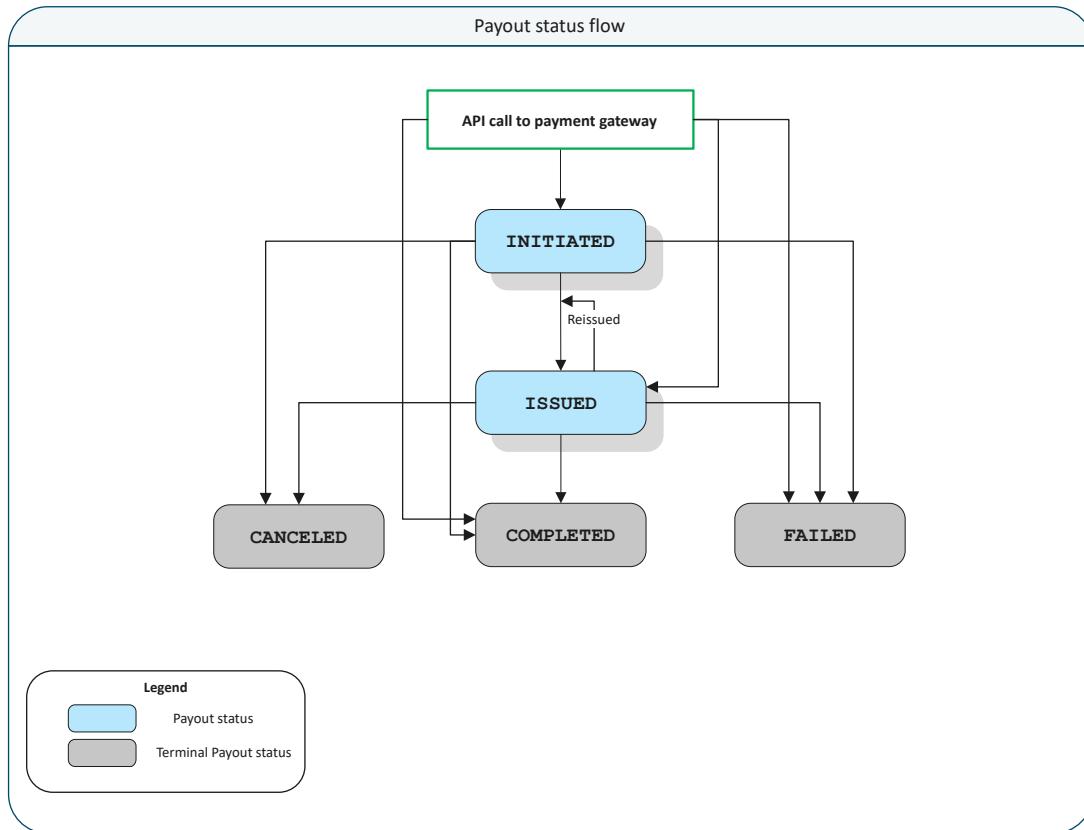
Generally, most payouts will proceed from `INITIATED` directly to `COMPLETED`, or, from `INITIATED` to `ISSUED` to `COMPLETED`, if all processes are successful.

The following table maps payout status in the payment gateway to check status in ClaimCenter according to the `TransactionStatusToInstantPaymentStatusMapper` property in the `InstantCheckUtil.gs` class in the base application:

Payout Status	Check Status	Description
INITIATED	Requested	<p>Instant check has been created in ClaimCenter, the Instant Check Transport message queue triggered, and the payment gateway has initiated payment.</p> <p>Note: This status is optional as the payment gateway may proceed directly to <code>COMPLETED</code> (though this situation is rare and not recommended).</p>
ISSUED	Issued	<p>Update from the payment gateway that a physical check or debit card was issued, or, for electronic payouts (such as PayPal), that the electronic payout was sent. Payment gateway provides other information such as check number and issued date, if available.</p> <p>Note: This status is optional as it is up to the payment gateway to determine if this status applies to payout processing. The gateway may ignore this status and proceed directly to <code>COMPLETED</code>.</p> <p>Note: Like other checks, an instant check can be reissued.</p>
COMPLETED	Cleared	<p>Instant payout is complete in the payment gateway. Payee verification and funds disbursement is successful.</p> <p>Note: This status means the money has been successfully received by the payee, and no errors or issues occurred. In some cases, such as an EFT disbursements by the gateway, this may take up to a few days to confirm a <code>COMPLETED</code> status.</p>
CANCELED	Stopped	<p>Instant check is stopped in ClaimCenter. As a result of this action in ClaimCenter, an API call is made to change the payout status to <code>CANCELED</code>. An attempt to halt processing of the payout is sent to the payment gateway.</p>
FAILED	Stopped (only if status is Pending Stop), otherwise no mapping.	<p>Instant payout failed in the payment gateway. In ClaimCenter, an error is logged, and an activity is created and assigned to the user that created the initial check. Issued date is updated if available. To attempt another payout, in ClaimCenter, you can clone the failed check, correct payee information, and issue</p>

Payout Status	Check Status	Description
		<p>another copy. Another possible option in ClaimCenter is to void the failed check.</p> <p>Note: No check status is updated in ClaimCenter except to move the check to Stopped if the status is currently Pending Stop.</p>

The following diagram shows the payout status flow from start to finish in the payment gateway:



When payout reaches **CANCELED**, **COMPLETED**, and **FAILED**, these are considered terminal events in the payment gateway. This means that payment gateway processing is finished and no further status updates are expected.

Note: The path from one payout status to another is linear from top to bottom as depicted. For example, it is not possible for a payout to go from **ISSUED** back to **INITIATED**. However, statuses can be skipped. For example, a payout can go from **INITIATED** to **COMPLETED**, or go directly to **FAILED**.

See also

- “Voiding, stopping, denying, and reissuing checks” on page 430
- ClaimCenter Application Guide*

Payout events that do not affect check status

There are cases where responses from the payment gateway do not alter the check's status in ClaimCenter, such as a message from the gateway that an SMS (text) message was sent to the payee, or, that the gateway has verified the payee's identity. In these cases, the payout status is returned to ClaimCenter as `null`. Lastly, payment gateway statuses are not limited to those in the previous table and may include more granular statuses or events.

Payment transaction integration

The following table includes the status codes and meanings for payments. The last two columns indicate whether the status exists for standard checks or manual checks.

Payment status	Meaning	Standard	Manual
null	An internal initial status.	Yes	Yes
pendingapproval	The associated check was saved in ClaimCenter but is not yet approved. In the base configuration, ClaimCenter auto-approves manual checks. You can customize this behavior with approval rules.	Yes	Yes
rejected	The approver did not approve the associated check.	Yes	Yes
awaitingsubmission	The associated check is held in this status until the date reaches the check's issuance date and a background processing task prepares the check for submission.	Yes	--
submitting	The payment is ready to be sent to an external system. This state corresponds to requesting status for an associated standard check or notifying status for an associated manual check.	Yes	Yes
submitted	The payment was sent to an external system. This state corresponds to requested, issued, or cleared status for an associated standard check or issued or cleared status for an associated manual check.	Yes	Yes
pendingvoid	Void request for the associated check is ready to be sent to an external system.	Yes	Yes
pendingstop	A stop request for the associated check is ready to be sent to an external system.	Yes	Yes
pendingtransfer	The associated check is ready to be transferred to another claim and is ready to send appropriate notification to an external system.	Yes	Yes
pendingrecode	The payment recode notification is ready to be sent to an external system.	Yes	Yes
recoded	The payment recode notification was sent to an external system.	Yes	Yes
transferred	The check was transferred to another claim.	Yes	Yes
voided	The associated check was voided.	Yes	Yes
stopped	The associated check was stopped.	Yes	Yes

To detect new or changed payments, you can write event business rules that listen for the `PaymentStatusChanged` event and check for changes to the `payment.Status` property. The following table includes the possible status code transitions and how this transition can occur. The rightmost two columns indicate whether the transition exists for associated standard check or manual checks.

Original payment status	Changed status	How the status change occurs	Standard	Manual
null	pendingapproval	The user created a new check and it requires approval. In the base configuration, ClaimCenter auto-approves manual checks. You can customize this behavior with approval rules.	Yes	Yes

Original payment status	Changed status	How the status change occurs	Standard	Manual
	awaitingsubmission	The user creates a standard check, and approval rules specify that the check does not require approval.	Yes	--
	submitting	A user created a manual check, and approval rules specify that the check does not require approval. In the base configuration, ClaimCenter auto-approves manual checks. You can customize this behavior with approval rules.	--	Yes
pendingapproval	awaitingsubmission	The approver approved the associated standard check, and it does not need further approval.	Yes	--
	rejected	The approver rejected the associated check.	Yes	Yes
	submitting	The approver approved the associated manual check.	--	Yes
	<i>object deleted</i>	A user deleted the object in the user interface.	Yes	Yes
rejected	awaitingsubmission	A user edited the associated check and approval rules specify that the check does not require approval.	Yes	--
	pendingapproval	A user edited the associated check, and approval rules specify that the check requires approval.	Yes	Yes
	submitting	A user edited the associated manual check, and approval rules specify that the check does not require approval.	--	Yes
	<i>object deleted</i>	A user deleted the object in the user interface.	Yes	Yes
awaitingsubmission	submitting	Automatic escalation batch process.	Yes	--
	<i>object deleted</i>	The user deleted the object in the user interface.	Yes	--
	pendingapproval	The user edited the associated check, and one or more payments on the check changes, and the check requires reapproval after that change.	Yes	--
submitting	submitted	<p>ClaimCenter received an acknowledgment for the associated message for the submitting status. This transition requires you to call <code>check.acknowledgeSubmission</code> in your message ACK code in your messaging plugins.</p> <p>The check and the associated payment are updated with new statuses. Also, the status of associated payments for onset or offset payments created for a recode or transfer are automatically updated to match the new payment status.</p>	Yes	Yes
	pendingvoid	<p>Either of the following events occurred:</p> <ul style="list-style-type: none"> • A user attempted to void an associated check in the user interface. • Some code called the <code>voidCheck</code> method—either the SOAP API method or the identically named domain method—on a check. 	Yes	Yes
	pendingstop	<p>Either of the following events occurred:</p> <ul style="list-style-type: none"> • A user attempted to stop an associated check in the user interface. • Some code called the <code>stopCheck</code> method—either the SOAP API method or the identically named domain method—on a check. 	Yes	Yes

Original payment status	Changed status	How the status change occurs	Standard	Manual
	pendingstransfer	A user requested a check transfer to another claim for the associated check.	Yes	Yes
	pendingrecode	A user requested a payment recoding. Unlike most other payment status transitions, a recode is about the payment, not simply a mirror or translation of an associated check status.	Yes	Yes
submitted	pendingvoid	<p>Either of the following events occurred:</p> <ul style="list-style-type: none"> • A user attempted to void an associated check in the user interface. • Some code called the <code>voidCheck</code> method—either the SOAP API method or the identically named domain method—on a check. 	Yes	Yes
	pendingstop	<p>Either of the following events occurred:</p> <ul style="list-style-type: none"> • A user attempted to stop an associated check in the user interface. • Some code called the <code>stopCheck</code> method—either the SOAP API or the identically named domain method—on a check. 	Yes	Yes
	pendingstransfer	A user requested a check transfer to another claim.	Yes	Yes
	pendingrecode	A user requested a payment recoding. Unlike most other payment status transitions, a recode is primarily about the payment, not simply a mirror or translation of an associated check status.	Yes	Yes
pendingvoid	voided	SOAP API for a check status change that indicates successful voiding.	Yes	Yes
	stopped	SOAP API for a check status change that indicates that the check was stopped. The void was unsuccessful.	Yes	Yes
	submitted	SOAP API for a check status change that indicates that the check was submitted. The void was unsuccessful. This happens if the associated check's status changed to issued or cleared.	Yes	Yes
pendingstop	stopped	SOAP API for a check status change that indicates stopping.	Yes	Yes
	voided	SOAP API for a check status change that indicates that the check was voided. The stop was unsuccessful.	Yes	Yes
	submitted	SOAP API for a check status change that indicates that the check was submitted. The stop was unsuccessful. This happens if the associated check's status changed to issued or cleared.	Yes	Yes
pendingtransfer	transferred	<p>ClaimCenter received an acknowledgment for the associated message for the pendingtransfer status. This transition requires you to call <code>check.acknowledgeTransfer</code> in your message ACK code in your messaging plugins.</p> <p>The check and the associated payment are updated to the transferred status.</p>	Yes	Yes
pendingrecode	recoded	ClaimCenter received an acknowledgment for the associated message for the pendingrecode status. This transition requires you to call <code>payment</code> .	Yes	Yes

Original payment status	Changed status	How the status change occurs	Standard	Manual
		acknowledgeRecode in your message Ack code in your messaging plugins.		
recoded	pendingvoid	<p>Either of the following events occurred:</p> <ul style="list-style-type: none"> • A user attempted to void an associated check in the user interface. • Some code called the <code>voidCheck</code> method—either the SOAP API or the identically named domain method—on a check. 	Yes	Yes
	pendingstop	<p>Either of the following events occurred:</p> <ul style="list-style-type: none"> • A user attempted to void an associated check in the user interface. • Some code called the <code>stopCheck</code> method—either the SOAP API or the identically named domain method—on a check. 	Yes	Yes
transferred	No transitions possible	Not applicable.	--	--
voided	submitted	SOAP API for a check status change that indicates that the check issued. The void request failed.	Yes	Yes
stopped	submitted	SOAP API for a check status change that indicates that the check issued. The stop request failed.	Yes	Yes

Required message acknowledgments for payments

Message acknowledgments trigger certain automatic status transitions. In these cases, your business rules must generate a message for these events. The messaging plugin representing the receiving system must acknowledge the message for the transition to occur. For payments, acknowledgment-driven status transitions apply in the following cases:

- Payment status: `pendingrecode` → `recoded`
- Payment status: `submitting` → `submitted`
- Payment status: `pendingtransfer` → `transfer`

Integration events for payment recoding

ClaimCenter supports payment recoding, which is a type of accounting change that does not issue a new check. These offsets connect with the original check. Every payment has a reserve line, which is a ClaimCenter encapsulation of a specific claim, a specific exposure, a specific cost type, and a specific cost category. This reserve line coding of payments provides critical data for accounting departments to track an insurance company's many payments. Insurance companies often generate a check, record the payment, and later recode the payment to maintain the payment amount but reassign it to a different reserve line for accounting purposes.

Within a reserve line definition, you can set the cost type component or the cost category component to `null` as appropriate to the exposure component. The claim is the only non-nullable component.

During payment recoding, ClaimCenter generates the following events.

- Events on the original payment that indicate that the original payment recoded.
- Events on a new offset payment, a Payment that effectively negates the original payment.
- Events on a new onset payment, a Payment that encodes a new payment in the correct reserve line.

Payment properties related to new offset and onset payments

To work with new offset and onset payments, you can use the following properties on the `Payment` entity instance.

Offsets

For a transferred or recoded payment, the value is its offsetting payment.

Onset

For a transferred or recoded payment, the value is the new onset payment on the target (destination) claim.

OnsetOriginPayment

If a payment is an onset payment resulting from a check transfer or a payment recoding, the value is the transferred or recoded payment for which the payment is an onset.

Onsets

For a recoded payment, the value is an array of `TransactionOnset` entity instances. Otherwise, the array has no entries. The `Onset` property of the transaction onset is a new onset payment for the target (destination) claim.

PaymentBeingOffset

If the payment is an offset, the value is the payment being offset by this offset payment. If the payment is not an offset, the value is `null`.

RecodingOffset

If the payment is an offset created for a recoded payment, the value is `true`. Otherwise, the value is `false`.

RecodingOnset

If the payment is an onset created for a recoded payment, the value is `true`. Otherwise, the value is `false`.

In the processing that your messaging destination performs for the `pendingrecode` message, the message acknowledgments must call the method `Payment.acknowledgeRecode` on the original payment.

Calling this financials acknowledgment API changes the status on the original payment from `pendingrecode` to `recoded`. It also updates the offset and onset payments to the `submitted` status.

Call this method only for recoding a payment, not for a regular payment associated with a check.

Events triggered during payment recoding

The following table lists events that trigger during recoding. The order in which events are triggered is not necessarily the order listed in this table. The event ordering is the event name ordering within each messaging destination.

Root object	Event name	Meaning in this context
Original payment	PaymentChanged	The payment <code>Status</code> property changed. You can catch this event, but you can also use the special <code>PaymentStatusChanged</code> event listed later in this table.
	PaymentStatusChanged	The payment <code>Status</code> property changed, specifically to <code>pendingrecode</code> . Use the <code>Payment</code> properties <code>Onset</code> and <code>Offsets</code> to get the associated payments.
New offset payment	PaymentAdded	New offset payment. Within business rules, check <code>payment.RecodingOffset</code> to determine if this is the offset payment. Use <code>Payment.PaymentBeingOffset</code> to get the original payment.
	PaymentStatusChanged	New financial transaction entities also receive a status changed event after their creation.
New onset payment	PaymentAdded	New onset payment. Within business rules, check <code>payment.RecodingOnset</code> to determine if this is the onset payment. Use <code>payment.OnsetOriginPayment</code> to get the original payment.
	PaymentStatusChanged	New financial transaction entities also receive a status changed event after their creation.

Integration payment events for check transfer

ClaimCenter users can transfer a check from one claim to another. Catch this event by listening for the `CheckStatusChanged` event and check for the status value `pendingtransfer`. Similarly, you can listen for the `PaymentStatusChanged` event and check for the status value `pendingtransfer`.

At the time that the status change event triggers, all the status changes and new payment creation has finished and must be considered final. To get values about the pre-transfer and post-transfer values and entities, use the properties and methods on `Check` and `Payment` entity types.

Your Event Fired rules that handle check transfers can use methods and properties on the `Payment` entity instance to handle either `CheckStatusChanged` events or `PaymentStatusChanged` events.

After the destination's messaging plugins send the associated message for the `pendingtransfer` event, the plugins get an acknowledgment back from the external system. After acknowledging the message itself, your messaging plugins must call the `Check.acknowledgeTransfer` method to complete this status transition. If you use that method, the status on the original payment changes from `pendingtransfer` to `transferred`. This change also updates the offset and onset payments to the submitted status.

Payment integration properties for event fired rules

The following table lists payment object properties that you use in Event Fired rules to test what type of payment generated the event. Use these properties to determine how to generate new messages to external systems.

In some cases, if voiding and recoding a payment, ClaimCenter creates multiple `Payment` entities. You must be careful in your rules that handle payment events such as `PaymentAdded` and `PaymentStatusChanged` so that you do not cause duplicate messages. To avoid this problem, check whether a payment is an onset payment or an offset payment.

Property to read	Value
<code>Check.TransferredCheck</code>	If the check is the result of a transfer, the original check. Otherwise, the value is <code>null</code> .
<code>Check.TransferredToCheck</code>	If the check transferred, the new check that was created on the target (destination) claim. Otherwise, the value is <code>null</code> .
<code>Payment.Offsets</code>	For a transferred or recoded payment, an array of a single <code>TransactionOffset</code> entity instance. Otherwise, the array has no entries. The <code>Offset</code> property of the transaction offset is the offset payment for this payment. For best practice, check that this array has a single entry.
<code>Payment.Onset</code>	For a transferred payment, the new onset payment on the target (destination) claim. Otherwise, the value is <code>null</code> . Do not use this property to access an onset payment for a recoded payment because a recoded payment can be split into multiple payments on different reserve lines. Reading this property on a split recoded payment causes an exception.
<code>Payment.OnsetOriginPayment</code>	If a payment is an onset payment resulting from a check transfer or a payment recoding, the transferred or recoded payment for which it is an onset. If the payment is not an onset, the value is <code>null</code> .
<code>Payment.Onsets</code>	For a recoded payment, an array of <code>TransactionOnset</code> entity instances. Otherwise, the array has no entries. The <code>Onset</code> property of the transaction onset is a new onset payment for the target (destination) claim.
<code>Payment.PaymentBeingOffset</code>	For a payment that is the offset for a transferred payment or a recoded payment, the transferred or recoded payment being offset. If the payment is not an offset, the value is <code>null</code> .
<code>Payment.RecodingOffset</code>	If this payment is an offset payment that is the result of a payment recoding, <code>true</code> . Otherwise, the value is <code>false</code> .

Property to read	Value
Payment.RecodingOnset	If this payment is an onset payment that is the result of a payment recoding, true. Otherwise, the value is false.
Payment.TransferOffset	If the payment is the offsetting payment for a transferred payment, true. Otherwise, the value is false.
Payment.TransferOnset	If the payment is the new payment created on the target claim for a transferred payment, true. Otherwise, the value is false.
Payment.Transferred	If this payment is transferring or transferred, true. Otherwise, the value is false. The value is true if the status is pendingtransfer or transferred.

Recovery reserve transaction integration

Recovery reserve transactions can have the following statuses.

Recovery reserve status	Description
null	An internal initial status.
pendingapproval	The associated recovery reserve was saved in ClaimCenter but is not yet approved.
rejected	The approver did not approve the associated recovery reserve.
submitting	The approver approved the recovery reserve and it is ready to be send to an external system.
submitted	The recovery reserve was sent to an external system and acknowledged.

To detect new or changed recovery reserves, you can write event business rules that listen for the `RecoveryReserveStatusChanged` event and check for changes to the `recoveryreserve.status` property. The following table includes the possible status code transitions and how each transition can occur.

Original recovery reserve status	Changed status	How the status change occurs
null	submitting	New recovery reserves make this transition if they are auto-approved. This behavior in the base configuration of ClaimCenter can be customized with approval rules.
	pendingapproval	New recovery reserves make this transition if they require approval. This behavior is not provided in the base configuration of ClaimCenter. You can customize this behavior with approval rules.
pendingapproval	submitting	The approver approved the recovery reserve and no additional approval is necessary.
	rejected	The approver rejected the recovery reserve.
	Object deleted	A user deleted the recovery reserve in the user interface.
rejected	Object deleted	A user deleted the recovery reserve in the user interface.
submitting	submitted	ClaimCenter received an acknowledgment for the associated message for the submitting status. This transition requires you to call <code>transaction.acknowledgeSubmission</code> in your message ACK code in your messaging plugins.
submitted	No transitions possible	Not applicable

Required message acknowledgments for recovery reserves

Message acknowledgments in some cases trigger certain automatic status transitions. In these cases, your business rules must generate a message for these events. The messaging plugin representing the messaging transport must call a financials acknowledgment API for the status transition to complete. For recovery reserves, acknowledgment-driven status transitions apply for a recovery reserve status change from `submitting` to `submitted`.

Recovery transaction integration

The following table lists the status codes and descriptions for recovery transactions.

Recovery status	Description
null	An internal initial status.
submitting	The recovery was approved and ready is to send to an external system. In the base configuration of ClaimCenter, there is no approval step for recoveries, but this step can be customized with approval rules.
submitted	The recovery was sent to an external system and acknowledged.
pendingvoid	The check void request is ready to be sent to an external system.
voided	The void request was sent and acknowledged.
pendingapproval	The recovery was saved in ClaimCenter but is not yet approved. In the base configuration of ClaimCenter, there is no approval step for recoveries, but this step can be customized with approval rules.
rejected	The approver did not approve the check request.

To detect new or changed recoveries, you can write event business rules that listen for the `RecoveryStatusChanged` event and check for changes to the `recovery.Status` property. The following table includes the possible status code transitions and how this transition can occur.

Original recovery status	Changed status	How the status change occurs
null	submitting	New recoveries make this transition if they are auto-approved. This behavior is supported in the base configuration of ClaimCenter, and it can be customized with approval rules.
	pendingapproval	New recoveries make this transition if they require approval. This behavior is not supported in the base configuration of ClaimCenter, but it can be customized with approval rules.
submitting	submitted	ClaimCenter received an acknowledgment for the associated message for the <code>submitting</code> status. This transition requires you to call <code>recovery.acknowledgeSubmission</code> in your message ACK code in your messaging plugins.
	pendingvoid	A user attempted to void a recovery.
submitted	pendingvoid	A user attempted to void a recovery.
pendingvoid	voided	ClaimCenter received an acknowledgment for the associated message for the <code>pendingvoid</code> status. This transition requires you to call <code>recovery.acknowledgeVoid</code> in your message ACK code in your messaging plugins.
voided	No transitions possible	Not applicable
pendingapproval	submitting	The approver approved the recovery and no additional approval is necessary.
	rejected	The approver rejected the recovery
rejected	Object deleted	A user deleted the recovery in the user interface.

Required message acknowledgments for recoveries

Message acknowledgments in some cases trigger certain automatic status transitions. In these cases, your business rules must generate a message for these events. Immediately after acknowledging the financials message itself, your messaging plugins must call a financials acknowledgment API to complete the status transition. For recoveries, acknowledgment-driven status transitions apply in the following cases.

- Recovery status: `submitting` → `submitted`
- Recovery status: `pendingvoid` → `voided`

Denying recoveries

Denial of a recovery supports downstream systems that may have additional criteria for accepting the recovery that cannot be incorporated into standard ClaimCenter business rules. An example for recovery is an invalid payer because the payer is on a watch list. To deny a recovery, the Recovery object must be in the `submitting` or `submitted` status.

To deny a recovery from web services, call the `ClaimFinancialsAPI` method `denyRecovery`. It takes a recovery public ID.

To deny a recovery from Gosu, such as from a messaging plugin, call the method directly on the Recovery object with no arguments.

```
recovery.denyRecovery()
```

Reserve transaction integration

To detect new or changed reserves, you can write event business rules that listen for the `ReserveStatusChanged` event and check for changes to the `reserve.status` property. The following table lists the status codes for reserve transactions.

Reserve status	Description
<code>null</code>	An internal initial status.
<code>submitting</code>	The approver approved the reserve and it ready is to be sent to an external system. Or it is an offset (positive or negative) and the associated payment moved to the <code>submitting</code> status.
<code>submitted</code>	The reserve was sent to an external system and was acknowledged.
<code>pendingapproval</code>	The reserve was saved in ClaimCenter and is pending approval.
<code>rejected</code>	The approver did not approve this reserve.
<code>awaitingsubmission</code>	The reserve is either a non-eroding offsetting reserve or a zeroing offsetting reserve for a payment. The payment that it is offsetting is waiting for submission.

To detect new or changed reserves, you can write event business rules that listen for the `ReserveStatusChanged` event and check for changes to the `reserve.Status` property. The following table lists the possible status code transitions and describes how each transition can occur.

Original reserve status	Changed status	How the status change occurs
<code>null</code>	<code>submitting</code>	A new reserve makes this transition if approval rules and authority limits indicate that the reserve does not require approval and is part of a reserve set (not a check set).
	<code>pendingapproval</code>	A new reserves makes this transition if approval rules and authority limits indicate that the reserve requires approval.

Original reserve status	Changed status	How the status change occurs
	awaitingsubmission	The reserve acts either as a non-eroding offsetting reserve or as a zeroing offsetting reserve for a payment.
submitting	submitted	ClaimCenter received an acknowledgment for the associated message for the submitting status. This transition requires you to call <code>reserve.acknowledgeSubmission</code> in your message ACK code in your messaging plugins.
submitted	No transition possible	Not applicable
pendingapproval	submitting	The approver approves the recovery and it does not require further approval.
	rejected	The approver rejected the reserve.
	Object deleted	A user deleted the recovery in the user interface.
rejected	Object deleted	A user deleted the recovery in the user interface.
awaitingsubmission	submitting	The reserve is either a non-eroding offsetting reserve or a zeroing offsetting reserve, and the payment that it offsets transitions to the submitting state.
	Object deleted	<p>Deletion automatically occurs if the associated payment is deleted or if a reserve payment change makes the offsetting reserve unnecessary. Examples of the latter case include the following:</p> <ul style="list-style-type: none"> • A non-eroding payment changes to an eroding payment. • A current day payment that exceeds reserves has its owning check rescheduled to be a future check. • The payment amount changes, and either it no longer exceeds reserves or it exceeds reserves by a different amount.

Required message acknowledgments for reserves

Message acknowledgments in some cases trigger certain automatic status transitions. In these cases, your business rules must generate a message for these events. Immediately after acknowledging the financials message itself, your messaging plugins must call a financials acknowledgment API to complete the status transition.

For reserves, acknowledgment-driven status transitions apply for a recovery reserve status change from `submitting` to `submitted`.

Bulk invoice integration

Use bulk invoices to create and submit a list of payments for many claims quickly without using the **New Check** wizard user interface.

This can simplify an insurance company's relationship with some large vendors or partners. For example, suppose an auto insurance company insures a large rental car company. At the end of the month, the rental car company could invoice the insurance company for all payments owed. The insurance company could use the ClaimCenter user interface to quickly enter payments by claim number, validate payment amounts and claim numbers, then finally submit payments for approval and processing.

The user interface provides rows of text properties for quick entry of each payment on the invoice. Because there is lots of data, users can make and save draft versions of an invoice that persist to the ClaimCenter database before submitting the final version. This is a feature that the regular check wizard in ClaimCenter does not have.

ClaimCenter also provides an optional web service (`BulkInvoiceAPI`) that an external system can use to submit bulk invoices to ClaimCenter. This avoids entirely the need to use the web user interface to enter and submit bulk invoices. Therefore, bulk invoices and all their associated line items can be programmatically imported using this API.

Bulk invoice validation

If you enter data into the **Bulk Invoice** screen and you are ready to complete the bulk invoice, click the **Submit** button to validate the invoice. This validate step ensures that the bulk invoice does not violate your business logic requirements for bulk invoices.

Bulk invoice validation is separate from bulk invoice approval. In general, validation determines whether the data appears to be correct, such as checking whether only certain vendors can submit any bulk invoices. In contrast, submitted invoices go through the approval process with business rules or human approvers to approve or deny an invoice before final processing.

Some types of validation happen automatically. Do not duplicate the validation logic for the following items in your validation plugin (`IBulkInvoiceValidationPlugin`).

- Basic claim number validity – In the user interface, the user interface code checks basic claim number validity. The `BulkInvoiceAPI` web service method `addItemsToInvoice` verifies this automatically.
- Claim and exposure validation levels – In the user interface, the user interface code checks claim and exposure validation levels to see if they reach the validation level Ability to Pay. The `BulkInvoiceAPI` web service method `addItemsToInvoice` verifies this automatically.
- Check if payments exceed reserves – In the user interface, the user interface code checks whether payments exceed reserves for those claims and exposures. The `BulkInvoiceAPI` web service method `submitBulkInvoice` verifies this automatically.

After a bulk invoice passes validation successfully, users can submit the invoice for approval and final processing. Final processing includes system-level validation tests on each approved invoice item, followed by creation of an individual check for each invoice item. ClaimCenter creates these checks as placeholders for accounting purposes on each affected claim. Bulk invoices have their own approval rule set called Bulk Invoice Approval.

A plugin interface called `IBulkInvoiceValidationPlugin` controls bulk invoice validation.

ClaimCenter provides a demonstration bulk invoice validation plugin implementation in Gosu. Use this plugin implementation for testing purposes only. To can access the example Gosu plugin, in the **Project** window of Guidewire Studio, navigate to **configuration > gsrc > gw > plugin > bulkinvoice.impl**, and open the file `SampleBulkInvoiceValidationPlugin`.

Your own implementation of the bulk invoice validation plugin must implement the `IBulkInvoiceValidationPlugin` interface, which contains one method.

```
public BIViolationAlert[] validateBulkInvoice(BulkInvoice invoice)
```

In this method, run your own validations on the `BulkInvoice` parameter. Call scriptable domain methods of the bulk invoice entity to access or calculate whatever is necessary to test validity.

- If all tests pass, this method must return `null` or an empty array.
- If one or more tests fail, construct one or more instances of the error message entity `BIViolationAlert` and return all the instances in a single array. Each instance must have an alert type and a message indicating the reason for the test failure. The default type is `Unspecified`. However, this method can use a more specific type in the extendible typelist `BIViolationAlertType`. The user interface PCF files can use the alert type of any alerts to generate different behaviors compared to the alert type `Unspecified`.

If ClaimCenter receives null or an empty array as the return value for the `validateBulkInvoice` method, ClaimCenter marks the bulk invoice as Valid. Otherwise, ClaimCenter marks the bulk invoice as Not Valid and commits the returned `BIValidationAlerts` to the database so the **Bulk Invoice** details page can display alerts.

If you do not register a bulk invoice validation plugin and you click the **Validate** button in the bulk invoice user interface, ClaimCenter immediately marks the bulk invoice as Valid.

See also

- *Gosu Rules Guide*

Bulk invoice web service APIs

The ClaimCenter `BulkInvoiceAPI` web service allows external systems to submit bulk invoices directly. For example, an associated rental car company could directly submit bulk invoices to ClaimCenter from other systems using these web service APIs. `BulkInvoiceAPI` methods can create and submit `BulkInvoice` objects, as well as add, update, and delete `BulkInvoiceItem` objects. Additionally, other methods do things like void, stop, and place holds on a bulk invoice.

This web service uses Data Transfer Objects to represent entity data. In this case, the main DTO objects are Gosu classes called `BulkInvoiceDTO` and `BulkInvoiceItemDTO`.

Many returned entities contain foreign key IDs rather than direct subobject links to associated entities. For example, calling any of the methods with names that begin with `getItems` return an array of `BulkInvoiceItemDTO` DTO objects. Each bulk invoice item DTO has a `BulkInvoiceID` property, which is the public ID of the `BulkInvoice` entity instance that contains it. To get a reference to the `BulkInvoice` entity instance to read its properties, make a separate call to `BulkInvoiceAPI` method `getBulkInvoice`.

Similarly, each bulk invoice item links to the related claim and exposure with public ID values in the properties `ClaimID` and `ExposureID`.

Adding items to an invoice

To add items to an invoice, from an external system call the `BulkInvoiceAPI` method `addItemsToInvoice`. The method takes as arguments a `String` invoice public ID and an array of `BulkInvoiceItemDTO` objects. The method returns an array of public IDs for the newly-added `BulkInvoiceItem` objects.

The following `BulkInvoiceItemDTO` properties are required at minimum.

- `Amount`, but only if the property `BulkInvoice.SplitEqually` is set to `false`
- `ClaimID`
- `CostCategory`
- `CostType`
- `PaymentType`

If any `BulkInvoiceItemDTO` object has invalid data, the method throws an exception in which case no items are added to the invoice.

Getting a bulk invoice

To get a bulk invoice from an external system, call the `BulkInvoiceAPI` method `getBulkInvoice`. The method takes a `String` invoice public ID and returns a `BulkInvoiceDTO` object.

Getting invoice items

To get invoice items, there are multiple `BulkInvoiceAPI` methods you can call from an external system.

Method name	Description
getItemsForInvoice	Gets all invoice items on a specific invoice.
getItemsForInvoiceAndClaim	Gets invoice items on a specific invoice, filtered by a specific String claim public ID.
getItemsForInvoiceAndClaimAndAmount	Gets invoice items on a specific invoice, filtered by a specific String claim public ID and amount of type BigDecimal of the invoice items.

All these methods return an array of BulkInvoiceItemDTO objects.

Creating a bulk invoice

To create a bulk invoice from an external system, call the BulkInvoiceAPI method `createBulkInvoice`. Populate a BulkInvoiceDTO object and pass it as a method argument. The following BulkInvoiceDTO properties are required at minimum.

- The payee property `PayeeID` on the bulk invoice
- `BulkInvoiceTotal`, but only if the property `BulkInvoice.SplitEqually` is set to `true`

The new bulk invoice has the status `draft`.

Adding invoice items is optional. Each invoice item must have all the required properties mentioned earlier for the `addItemsToInvoice` method. Additionally, be aware that if the server is configured in multicurrency mode, ClaimCenter selects a default market rate for the property `TransToReportingExchangeRate`. If you wish to provide your own custom rates, provide appropriate values for properties `NewExchangeRate` and `NewExchangeRateDescription` on the corresponding BulkInvoiceDTO object.

Updating items on an invoice

To update invoice items on a bulk invoice, from an external system call the BulkInvoiceAPI method `updateItemsOnInvoice`. Pass an invoice public ID and an array of invoice item DTOs that represent items to change.

Each invoice item must at minimum have the `PublicID` property to identify the item to change. Additionally, set additional properties that you want to change, such as `Amount`. By default, all `null` property values are ignored. As a consequence, by default you cannot set a value to `null` using this web service method. However, this behavior is configurable. Go to the WS-I web service implementation class. At the end of the `updateItems` method, find the following code statement.

```
itemDTO.writeTo(itemFromDB)
```

Change the statement to the following invocation.

```
itemDTO.writeTo(itemFromDB, false)
```

After this change, all values are set on the entity instance, including `null` values.

If you make the change to reverse the behavior of `null` values, be careful to always populate all invoice items properties to prevent data loss.

Deleting items from an invoice

To delete invoice items on a bulk invoice from an external system, call the BulkInvoiceAPI method `deleteItemsFromInvoice`. Pass an invoice public ID and an array of invoice item public IDs for the items to delete.

Validating an invoice

To validate invoice items on a bulk invoice from an external system, call the BulkInvoiceAPI method `validateBulkInvoice`. Pass the invoice public ID as an argument. Any validation failure messages are returned in an

array of `BIValidationAlertDTO` objects. If successful, the method returns an empty array. ClaimCenter also attaches and persists validation errors with the bulk invoice. Refer to the property `BulkInvoice.ValidationAlerts` which contains an array of `BIValidationAlert` entity instances.

Submitting an invoice

To submit an invoice for approval from an external system, call the `BulkInvoiceAPI` method `submitBulkInvoice`. Pass the invoice public ID as an argument. You can only use this API for invoices with status of `Draft`, `Rejected`, `OnHold`, or `PendingItemValidation`. The invoice must have at least one `BulkInvoiceItem`.

Requesting downstream escalation for an invoice

To request downstream escalation for an invoice from an external system, call the `BulkInvoiceAPI` method `requestBulkInvoice`. Pass the invoice public ID as an argument. You can only use this API for invoices with status of `AwaitingSubmission`.

This API is for highly unusual situations only. Typically, ClaimCenter handles escalation automatically using the scheduled `bulkinvoicesescalation` batch process. Use this API only if you want to manually escalate a `BulkInvoice` without waiting for the batch process to run.

Voiding a bulk invoice

To void a bulk invoice from an external system, call the `BulkInvoiceAPI` method `voidBulkInvoice`. Pass the invoice public ID as an argument. You can only use this API for invoices with status of `Requesting`, `Requested`, `Issues`, `Cleared`, or `OnHold`. This method sets the invoice and all its items to status `PendingVoid`. The method also voids any related bulk checks.

Stopping a bulk invoice

To stop a bulk invoice from an external system, call the `BulkInvoiceAPI` method `stopBulkInvoice`. Pass the invoice public ID as an argument. You can only use this API for invoices with status of `Requesting`, `Requested`, `Issues`, or `OnHold`. This method sets the invoice and all its items to status `PendingStop`. The method also stops any related bulk checks.

Placing a downstream hold on an invoice

To place a hold on a bulk invoice from an external system, call the `BulkInvoiceAPI` method `placeDownstreamHoldOnInvoice`. Pass the invoice public ID as an argument. You can only use this API for invoices with status of `Requesting`, `Requested`. This method sets the invoice to status `OnHold`.

Updating the status of an invoice

To update invoice status from an external system, call the `BulkInvoiceAPI` method `updateBulkInvoiceStatus`. You can optionally update the check number and issue date.

Possible status transitions and requirements are listed below.

- **Issued** – old status must be `Requesting`, `Requested`, `PendingVoid`, `PendingStop`, `Voided`, or `Stopped`.
- **Cleared** – old status must be `Requesting`, `Requested`, `Issued`, `PendingVoid`, `PendingStop`
- **Voided** – old status must be `PendingVoid` or `PendingStop`
- **Stopped** – old status must be `PendingVoid` or `PendingStop`

The method accepts the following arguments.

- The invoice public ID
- Optional new check number (pass `null` to ignore)
- Optional new issue date of the bulk check (pass `null` to ignore)

- New status

The `BulkInvoiceAPI` web service method `updateBulkInvoiceStatus` updates the status of the placeholder checks. The status of each placeholder check is automatically updated when the status of the bulk invoice changes. For example, setting the status of the bulk invoice to `Issued` automatically sets the status of all placeholder checks to `Issued`.

Typical bulk invoice API usage

Suppose you want to create a new bulk invoice and submit it. The following is a typical workflow for your integration code.

1. Populate a new `BulkInvoiceDTO` object to represent the new bulk invoice.
2. Set the following properties on the `BulkInvoiceDTO` object.

`CheckNumber`

The check number

`RequestingUser`

The requesting user

`SplitEqually`

(Boolean) Whether to split items equally. If `SplitEqually` has the value `true`, you must also set the `BulkInvoiceTotal` property on the DTO.

`PublicID`

Bulk invoice public ID

`InvoiceNumber`

Invoice number

`InvoiceItems`

Invoice items

Guidewire recommends you add invoice items directly on the DTO when you create the bulk invoice DTO. It is not strictly required to add the invoice items at this step. You could choose to add the invoice items separately at a later step.

3. Carefully set the `PayeeID` property on the `BulkInvoiceDTO` object. Set `PayeeID` to the public ID for the payee contact entity. Using Bulk Invoices requires using a contact system and the payees must be in the contact system, such as `ContactManager`. For the `PayeeID` property, ClaimCenter needs the public ID of the contact. To safely get the public ID, perform the following actions.
 - a. Ensure that the contact is in the contact system. If you use `ContactManager`, use the `ContactManager` web services `ABContactAPI` to add the contact. Whether you are using an already-existing contact or creating a new one, remember to save the “address book link ID.” An address book link ID is the native address book ID for the contact. The address book link ID is different from the public ID.
 - b. Call the `ClaimCenter BulkInvoiceAPI` web service method `createContactByLinkId` which takes the link ID and returns a contact entity.
 - c. Get the public ID from the result of the `createContactByLinkId` method and set the invoice `PayeeID` property to that value.
4. Call the `createBulkInvoice` method with your `BulkInvoiceDTO` object to create a new draft unapproved bulk invoice in the database.
5. If you did not add the invoice items yet, use the `addItemsToInvoice` method to add bulk invoice items on the bulk invoice.
6. Validate the bulk invoice with the `validateBulkInvoice` method.
7. If there were no validation errors, call the `submitBulkInvoice` method to submit the bulk invoice.

8. After creating the bulk invoice, you can optionally call the `updateItems` or `updateBulkInvoiceStatus` method to change the existing data.

Post-submission actions on a bulk invoice

The bulk invoice life-cycle is not necessarily complete after it submits to a downstream system. Possible updates can occur either through the ClaimCenter user interface, or through web services.

Updating a bulk invoice to issued or cleared status

Typically, after paying the bulk payment associated with the invoice, you want to update the business status of the invoice to `issued` and then `cleared`. You can use multiple APIs to do this, depending on whether you are calling from an external system or from your own messaging plugins.

Placing a downstream system hold on a bulk invoice

Occasionally, there may be problems with a bulk invoice that can only be detected on your server after submission. For this reason, you can place a hold on a submitted bulk invoice. Do this with the `BulkInvoiceAPI` method `placeDownstreamHoldOnInvoice`. Refer to the “Java API Reference” documentation for more information about this method.

After a downstream hold is on an invoice, only the following actions are possible.

- You can void the bulk invoice.
- You can stop the bulk invoice.
- You can resubmit the bulk invoice, as is.

Stopping a bulk invoice

You can stop a bulk invoice if it is considered stoppable by the system. Stopping a bulk invoice has the following effects.

- The status of the invoice transitions to `pendingstop`.
- For every invoice item that has an associated check, the invoice item status changes to `pendingstop`.
- ClaimCenter stops all associated checks. This is the same effect as stopping the checks individually through the user interface.

Once a bulk invoice transitions to `pendingstop`, it can only transition to `stopped` status using a call to the `BulkInvoiceAPI` method `updateBulkInvoiceStatus`. Any attempt to use this method to transition a bulk invoice to `stopped` if it has not already stopped (through the user interface or the API) results in an error.

If the stop on the bulk invoice is unsuccessful, use the method `updateBulkInvoiceStatus` on the bulk invoice in Gosu or the web service `BulkInvoiceAPI`. Those methods transition the invoice back to `issued` or `cleared` status with the following effects.

- The status of the invoice transitions to either `issued` or `cleared` status respectively.
- Every bulk invoice item that has an associated check transitions back to `submitted` status.
- Every associated check is un-stopped. This is the same behavior as if a regular un-bulked check stops, but then the stop attempt fails.

The `BulkInvoiceAPI` web service method `updateBulkInvoiceStatus` updates the status of the placeholder checks. The status of each placeholder check is automatically updated when the status of the bulk invoice changes.

Voiding a bulk invoice

You can void a bulk invoice if it is considered voidable by the system. Voiding a bulk invoice has the following effects.

- The status of the invoice transitions to `pendingvoid`.

- Every invoice item that has an associated check transitions to status `pendingvoid`.
- ClaimCenter voids all associated checks. This has the same effect as voiding the checks individually through the user interface.

Once a bulk invoice transitions to `pendingvoid`, it can only transition to `voided` status using one of the following methods.

- A call from an external system to the `BulkInvoiceAPI` method `updateBulkInvoiceStatus`.
- A call from your messaging plugins to the bulk invoice domain method `updateBulkInvoiceStatus`.

In both cases, the bulk invoice must already be at the status `pendingvoid`. The `updateBulkInvoiceStatus` method (in either variant) throws an exception if the status is not `pendingvoid`.

If the call to `updateBulkInvoiceStatus` to void a bulk invoice fails for any reason, you can use the `updateBulkInvoiceStatus` method to transition the invoice to a `issued` or `cleared` status. This has the following effects.

- The status of the invoice transitions to either `issued` or `cleared` status respectively.
- Every bulk invoice item that has an associated check transitions back to `submitted` status.
- Every associated check is un-voided. This is the same behavior as if a regular un-bulked check voided but then the void attempt failed.

The `BulkInvoiceAPI` web service method `updateBulkInvoiceStatus` updates the status of the placeholder checks. The status of each placeholder check is automatically updated when the status of the bulk invoice changes.

Bulk invoice processing performance

Bulk invoices printed or recorded by hand on paper usually have few line items. However, for automated creation and processing of bulk invoices using web services, incoming electronic invoices might be large. For example, a large favored supplier (for rental cars, medical billing adjustment services, and so on) might have thousands of line items. To handle one such huge electronic invoice, they may need to create multiple bulk invoices in ClaimCenter. Converting extremely large bulk invoices into smaller ClaimCenter invoices might be necessary to create, fix, and finally submit the bulk invoices in a reasonable amount of time.

Bulk invoice processing performance varies greatly. Performance depends on factors that include, but are not limited to, the following components.

- Server hardware
- Rule Set code
- Server configuration settings

Before you deploy your final production configuration of your ClaimCenter server, you must test your ClaimCenter configuration using the same hardware as your production server. Use either the same physical hardware or an exact copy. You must test performance for submitting a Bulk Invoice. Experiment to determine your settings for the following settings.

- Maximum time per invoice – The acceptable amount of time for ClaimCenter to process and fully submit any bulk invoice. As part of submitting, ClaimCenter might mark some bulk invoice items as Not Valid. You must plan on allocating necessary time to manually edit some items to fix issues and resubmit bulk invoices.
- Maximum items per invoice – Based on the maximum amount of time per invoice, establish the maximum number of items on each bulk invoice. This value will depend on your particular combination of server hardware, rule code, and server configuration.

Depending on the volume of invoices, you might need to design special procedures to maximize performance from large vendors. For example, you may want to do SOAP API handling and inevitable manual cleanup and resubmission on a dedicated ClaimCenter server in the cluster. That way, this process minimally affects regular ClaimCenter users.

It is important to distinguish Bulk Invoice Item processing from Bulk Invoice Escalation. The performance challenges are with bulk invoice item processing. Bulk invoice processing happens immediately after bulk invoice approval. The

bulk invoice item processing step creates a placeholder check for each bulk invoice item on each item's claim. By the end of the process, ClaimCenter creates and approves all checks and the bulk invoice has the status `AwaitingSubmission`.

The following steps occur for submitting a bulk invoice.

1. The user clicks the **Submit** button or an external system calls the `submitBulkInvoice` Soap API method.
2. If the bulk invoice passes Bulk Invoice Approval rules, proceed to the next step in this list. However, it may be that the Bulk Invoice Approval specifies additional approval is necessary. Eventually, a supervisor signs in and approves the Bulk Invoice Approval activity. At this point, no further approvals are necessary.
3. After final approval, bulk invoice item processing starts on that same machine.
4. If everything is valid, no checks require approval by TransactionSet Approval Rules, so the bulk invoice has the status `AwaitingSubmission`. ClaimCenter now creates the checks and runs all rules on them. This step is what takes so long during bulk invoice processing.
5. ClaimCenter starts bulk invoice processing by adding the bulk invoice to a single `TaskQueue` object per machine. Each task queue object processes each bulk invoice and all its items in order in a single thread. This process starts on the same machine on which Bulk Invoice Approval rules run. This thread is not a ClaimCenter batch process. This is a single thread and not a batch process. Because of this, Guidewire recommends for large vendors to perform bulk invoice operations on a separate machine. This includes both the web service APIs and the user actions.

Whether bulk invoice processing begins due to the user interface **Submit** button or the resolution of final approval, bulk invoice processing always takes place on that same machine. Thus, it is best to segregate these tasks to a separate machine so that it does not interfere with the CPU, disk, or network resources of regular application web users.
6. After the bulk invoice reaches the status `AwaitingSubmission`, the Bulk Invoice Escalation batch process finds the bulk invoice. Specifically, the escalation batch process finds bulk invoices in status `AwaitingSubmission` that reached their scheduled send date. The batch process moves any such bulk invoices to the status `Requesting`. The batch process sets the status of all the related bulk invoice items and their placeholder checks to the status `Requesting`. This status change happens mainly to update the status values and create an opportunity to create event messages in Event Fired rules. Like all message sending code, the message processing happens asynchronously as part of the send queue processing on a server with the `messaging` server role. The asynchronous messaging sending is typically not performance intensive.

Bulk invoice (top level entity) status transitions

The following table lists the status codes and meanings for bulk invoice transactions.

Bulk invoice status	Meaning
<code>null</code>	An internal initial status for a new <code>BulkInvoice</code> entity.
<code>draft</code>	Status for initial editing of an invoice, or after invalidation due to changes.
<code>inreview</code>	The bulk invoice requires approval and awaits action by the assigned approver.
<code>pendingbulkinvoiceitemvalidation</code>	The approver approved the bulk invoice. Individual bulk invoice items are now pending validation and undergoing final processing.
<code>invalidbulkinvoiceitems</code>	One or more invoice items failed validation or failed during check creation, or the check associated with one or more items was submitted for approval, and then the approver rejected it.
<code>awaitingsubmission</code>	The bulk invoice awaits submission to the downstream system.
<code>requesting</code>	The bulk invoice was queued for submission to the downstream system.

Bulk invoice status	Meaning
requested	The bulk invoice was sent successfully to the downstream system.
issued	The bulk invoice's associated bulk check was issued.
cleared	The bulk invoice's associated bulk check cleared.
rejected	The assigned approver rejected the bulk invoice.
pendingvoid	The bulk invoice was voided. Confirmation of the void is pending.
pendingstop	The bulk invoice was stopped. Confirmation of the void is pending.
voided	The bulk invoice was successfully voided.
stopped	The bulk invoice was successfully stopped.
onhold	The downstream system placed the bulk invoice on hold.

To detect new or changed reserves, you can write event business rules that listen for the `BulkInvoiceStatusChanged` event and check for changes to the `bulkinvoice.Status` property. The following table lists the possible status code transitions and how each transition can occur.

Original bulk invoice status	Changed status	How the status change occurs
null	draft	New BulkInvoice entities were created.
draft	inreview	<p>This transition occurs if both of the following conditions occur.</p> <ul style="list-style-type: none"> • A bulk invoice is submitted, either through the ClaimCenter user interface or by using the BulkInvoiceAPI web service method <code>submitBulkInvoice</code>. • The defined bulk invoice approval rules require it to escalate for approval.
	pendingbulkinvoiceitemvalidation	<p>This transition occurs if both of the following conditions occur.</p> <ul style="list-style-type: none"> • A bulk invoice is submitted, either through the ClaimCenter user interface or by using the BulkInvoiceAPI web service method <code>submitBulkInvoice</code>. • The defined bulk invoice approval rules do not require it to escalate for approval.
inreview	pendingbulkinvoiceitemvalidation	<p>This transition occurs if you approve a bulk invoice through the ClaimCenter user interface.</p>
	rejected	<p>This transition occurs if you reject a bulk invoice through the ClaimCenter user interface.</p>
	draft	<p>This transition occurs if you edit the bulk invoice in such a way that it no longer passes validation.</p>

Original bulk invoice status	Changed status	How the status change occurs
pendingbulkinvoiceitemvalidation	awaitingsubmission	This transition occurs after all approved bulk invoice items are successfully validated, and a check is automatically created for them, and the approver approves the check.
	invalidbulkinvoiceitems	This transition occurs if the processing of the bulk invoice's items complete and one or more items are now <code>notvalid</code> or the approver rejects one or more associated checks.
invalidbulkinvoiceitems	pendingbulkinvoiceitemvalidation	This transition occurs if the bulk invoice is resubmitted without first changing it to <code>draft</code> status again, and it does not require approval.
	draft	This transition occurs if you edit the bulk invoice in such a way that it no longer passes validation.
	inreview	This transition occurs if the bulk invoice is resubmitted, requires an approver, and its status is not changed to <code>draft</code> .
awaitingsubmission	draft	This transition occurs if you edit the bulk invoice and the changes you make cause it to no longer pass validation.
	requesting	This transition occurs for bulk invoices with a scheduled send date of the current day if the <code>bulkinvoicesescalation</code> batch process runs. The process runs either automatically at a scheduled time or manually from the command prompt. Alternatively, the SOAP APIs can escalate the bulk invoice.
requesting	requested	ClaimCenter received an acknowledgment for the associated message for the <code>submitting</code> status. This transition requires you to call <code>bulkInvoice.acknowledgeSubmission</code> in your message ACK code in your messaging plugins.
	issued	This transition can occur by calling the <code>updateBulkInvoiceStatus</code> method, either the SOAP API method or the domain method, to change the status to <code>issued</code> .
	cleared	This transition can occur by calling the <code>updateBulkInvoiceStatus</code> method, either the SOAP API method or the domain method, to change the status to <code>cleared</code> .
	pendingstop	This transition occurs if the bulk invoice is stopped, either through the ClaimCenter user interface or by using a call to the

Original bulk invoice status	Changed status	How the status change occurs
		BulkInvoiceAPI method <code>stopBulkInvoice</code> .
	<code>pendingvoid</code>	This transition occurs if the bulk invoice is voided, either through the ClaimCenter user interface or by using a call to the BulkInvoiceAPI method <code>voidBulkInvoice</code> .
	<code>onhold</code>	This transition occurs if the Bulk Invoice API places a downstream hold on the invoice by using a call to the <code>placeDownstreamHoldOnInvoice</code> method.
<code>requested</code>	<code>issued</code>	This transition can occur by calling the <code>updateBulkInvoiceStatus</code> method, either the SOAP API method or the domain method, to change the status to <code>issued</code> .
	<code>cleared</code>	This transition can occur by calling the <code>updateBulkInvoiceStatus</code> method, either the SOAP API method or the domain method, to change the status to <code>cleared</code> .
	<code>pendingstop</code>	This transition occurs if the bulk invoice is stopped, either through the ClaimCenter user interface or by using a call to the BulkInvoiceAPI method <code>stopBulkInvoice</code> .
	<code>pendingvoid</code>	This transition occurs if the bulk invoice is voided, either through the ClaimCenter user interface or by using a call to the BulkInvoiceAPI method <code>voidBulkInvoice</code> .
	<code>onhold</code>	This transition occurs if the bulk invoice API places a downstream hold on the invoice by using a call to the <code>placeDownstreamHoldOnInvoice</code> method.
<code>cleared</code>	<code>pendingvoid</code>	This transition occurs if the bulk invoice is voided, either through the ClaimCenter user interface or by using a call to the BulkInvoiceAPI method <code>voidBulkInvoice</code> .
<code>issued</code>	<code>cleared</code>	This transition can occur by calling the <code>updateBulkInvoiceStatus</code> method, either the SOAP API method or the domain method, to change the status to <code>cleared</code> .
	<code>pendingstop</code>	This transition occurs if the bulk invoice is stopped, either through the ClaimCenter user interface or by using a call to the BulkInvoiceAPI method <code>stopBulkInvoice</code> .

Original bulk invoice status	Changed status	How the status change occurs
	pendingvoid	This transition occurs if the bulk invoice is voided, either through the ClaimCenter user interface or by using a call to the BulkInvoiceAPI method <code>voidBulkInvoice</code> .
rejected	draft	This transition occurs if you edit the bulk invoice and your changes cause it to no longer pass validation.
	inreview	This transition occurs if both of the following conditions exist: <ul style="list-style-type: none"> The bulk invoice is resubmitted, either through the ClaimCenter user interface or by using the BulkInvoiceAPI method <code>submitBulkInvoice</code>. Bulk invoice approval rules require escalation for approval.
	pendingbulkinvoiceitemvalidation	This transition occurs if both of the following conditions exist: <ul style="list-style-type: none"> The bulk invoice is resubmitted, either through the ClaimCenter user interface or by using the BulkInvoiceAPI method <code>submitBulkInvoice</code>. The bulk invoice approval rules do not require it to escalate for approval.
pendingvoid	voided	This transition can occur by calling the <code>updateBulkInvoiceStatus</code> method, either the SOAP API method or the domain method, to change the status to <code>voided</code> .
	stopped	This transition can occur by calling the <code>updateBulkInvoiceStatus</code> method, either the SOAP API method or the domain method, to change the status to <code>stopped</code> .
	issued	This transition can occur by calling the <code>updateBulkInvoiceStatus</code> method, either the SOAP API method or the domain method, to change the status to <code>issued</code> . This status change means the void attempt was unsuccessful.
	cleared	This transition can occur by calling the <code>updateBulkInvoiceStatus</code> method, either the SOAP API method or the domain method, to change the status to <code>cleared</code> . This status change means that the void attempt was unsuccessful.
pendingstop	stopped	This transition can occur by calling the <code>updateBulkInvoiceStatus</code> method, either the SOAP API method or the

Original bulk invoice status	Changed status	How the status change occurs
	stopped	domain method, to change the status to stopped.
	voided	This transition can occur by calling the updateBulkInvoiceStatus method, either the SOAP API method or the domain method, to change the status to voided.
	issued	This transition can occur by calling the updateBulkInvoiceStatus method, either the SOAP API method or the domain method, to change the status to issued. This status change means that the stop attempt was unsuccessful.
	cleared	This transition can occur by calling the updateBulkInvoiceStatus method, either the SOAP API method or the domain method, to change the status to cleared. This status change means that the stop attempt was unsuccessful.
stopped	issued	This transition can occur by calling the updateBulkInvoiceStatus method, either the SOAP API method or the domain method, to change the status to issued. This status change means that the stop attempt was unsuccessful.
	cleared	This transition can occur by calling the updateBulkInvoiceStatus method, either the SOAP API method or the domain method, to change the status to cleared. This status change means that the stop attempt was unsuccessful.
voided	issued	This transition can occur by calling the updateBulkInvoiceStatus method, either the SOAP API method or the domain method, to change the status to issued. This status change means that the void attempt was unsuccessful.
	cleared	This transition can occur by calling the updateBulkInvoiceStatus method, either the SOAP API method or the domain method, to change the status to cleared. This status change means that the void attempt was unsuccessful.
onhold	pendingstop	This transition occurs if the bulk invoice is stopped, either through the ClaimCenter user interface or by using a call to the BulkInvoiceAPI method stopBulkInvoice.
	pendingvoid	This transition occurs if the bulk invoice is voided, either through the ClaimCenter user interface or by using a call to the BulkInvoiceAPI method voidBulkInvoice.

Original bulk invoice status	Changed status	How the status change occurs
	requesting	This transition occurs if the invoice is resubmitted, either through the ClaimCenter user interface or by using a call to the BulkInvoiceAPI method submitBulkInvoice.

Required message acknowledgments for bulk invoices

Message acknowledgments in some cases trigger certain automatic status transitions. In these cases, your business rules must generate a message for these events. Immediately after acknowledging the financials message itself, your messaging plugins must call a financials acknowledgment API to complete the status transition.

For bulk invoices, acknowledgment-driven status transitions apply if bulk invoice status is changing from `requesting` to `requested`.

After you acknowledge the submission message, the following updates occur.

1. The bulk invoice's status changes to `requested`.
2. The status of every invoice item with a current status of `submitting` changes to `submitted`.

Bulk invoice status changes using SOAP or domain method

Typically, after paying the bulk payment associated with the invoice, you want to update the business status of the invoice to `issued` and then `cleared`. You can use either of the following approaches.

- From an external system, call the SOAP API BulkInvoiceAPI method `submitBulkInvoice`. The BulkInvoiceAPI web service method `updateBulkInvoiceStatus` updates the status of the placeholder checks. The status of each placeholder check is automatically updated when the status of the bulk invoice changes.
- From your messaging code, call the bulk invoice domain method `updateBulkInvoiceStatus`. The BulkInvoiceAPI web service method `updateBulkInvoiceStatus` updates the status of the placeholder checks. The status of each placeholder check is automatically updated when the status of the bulk invoice changes.

This method allows you to set the bulk check number and the issue date on the bulk invoice.

The `updateBulkInvoiceStatus` method can make these changes.

- Update a bulk invoice from `requested` to `issued` status, which also sets the Issued Date. Typically this occurs after the bulk payment is made from the insurer to the vendor who submitted the bulk invoice.
- Update a bulk invoice to `cleared` status.
- Update a `pendingvoid` bulk invoice to `voided` status.
- Update a `pendingstop` bulk invoice to `stopped` status.
- Update a bulk invoice with a canceled status (the statuses `pendingstop`, `stopped`, `pendingvoid`, `voided`) to the `issued` or `cleared` status. This effectively means that the void or stop attempt failed.

From your messaging plugin, you can call the `bulkInvoice.updateBulkInvoiceStatus` method to change the status, and only after your messaging code acknowledges a message.

If a check is `pendingstop` or `pendingvoid` and the new status is `issued` or `cleared`, the status values of the check and its related payments change to the new value. This is true both for changes from SOAP APIs or domain methods. Next, ClaimCenter creates a warning activity. Next, ClaimCenter assigns the activity to the user who attempted to void or stop the check. This activity lets the user know that the check did not stop/void successfully. Finally, ClaimCenter generates any reserve that is necessary to keep open reserves from becoming negative.

Bulk invoice item status transitions

The following table lists the status codes and descriptions for bulk invoice item transactions.

Bulk invoice item status	Description
null	An internal initial status for a new BulkInvoiceItem entity not yet committed to the database.
draft	Bulk invoice item commits to the database but its owning invoice it is not yet submitted for approval. This is the initial status for every new invoice item.
approved	The approver approved the bulk invoice item and is valid for processing.
rejected	The assigned bulk invoice approver did not approve the bulk invoice item and ClaimCenter must not process it further.
submitting	Bulk invoice item is pending submission to the downstream system.
submitted	The bulk invoice item successfully submitted to the downstream system
inreview	Bulk invoice item requires action before it can be paid. This is the status during bulk invoice approval.
notvalid	The bulk invoice item failed validation or a problem occurred while creating its associated check.
pendingvoid	The bulk invoice item's owning bulk invoice voided and confirmation of the void is pending.
pendingstop	The bulk invoice item's owning bulk invoice stopped and confirmation of the void is pending.
voided	The bulk invoice item's owning bulk invoice successfully voided.
stopped	The bulk invoice item's owning bulk invoice successfully stopped.
pendingtransfer	The bulk invoice item's associated check is pending transfer.
transferred	The bulk invoice item's associated check successfully transferred.

To detect new or changed reserves, you can write event business rules that listen for the BulkInvoiceItemStatusChanged event and check for changes to the *bulkinvoiceitem.Status* property. The following table includes the possible status code transitions and how this transition can occur.

Original bulk invoice item status	Changed status	How the status change occurs
null	draft	A new BulkInvoiceItem
draft	approved	This transition occurs in the following cases: <ul style="list-style-type: none"> • Upon submit of the owning bulk invoice if the defined bulk invoice approval rules do not require escalation for approval. • If the bulk invoice is in review and the assigned approver either approves the entire bulk invoice, or does not mark the invoice item as rejected or inreview.
	rejected	This transition occurs if the bulk invoice is in review and either of the following is true: <ul style="list-style-type: none"> • The approving user rejects the entire bulk invoice. • The approving user approves the bulk invoice, but specifically marks this invoice item as rejected on the approval activity details worksheet.
	inreview	This transition occurs if the bulk invoice is in review and the approving user approves the bulk invoice, but specifically marks this invoice item as inreview.
approved	draft	This transition occurs if the bulk invoice item changes in such a way that the owning bulk invoice no longer passes validation.
	submitting	This transition occurs if the owning bulk invoice escalates for submission to the downstream system.

Original bulk invoice item status	Changed status	How the status change occurs
	notvalid	This transition occurs if an approved bulk invoice item fails a system validation check during the final processing phase. It also occurs if there is a problem creating and approving the associated check for a validated invoice item.
rejected	draft	This transition occurs if the bulk invoice item changes in such a way that the owning bulk invoice no longer passes validation.
submitting	submitted	ClaimCenter received an acknowledgment for the associated message for the submitting status. This transition requires you to call <code>bulkInvoice.acknowledgeSubmission</code> in your message ACK code in your messaging plugins.
	pendingvoid	This transition occurs if the owning bulk invoice stops either through the ClaimCenter user interface or by using a call to the BulkInvoiceAPI method <code>stopBulkInvoice</code> .
	pendingstop	This transition occurs if the owning bulk invoice voids either through the ClaimCenter user interface or by using a call to the BulkInvoiceAPI method <code>voidBulkInvoice</code> .
	pendingtransfer	This transition occurs if the check associated with the bulk invoice item transfers from the ClaimCenter user interface.
submitted	pendingvoid	This transition occurs if the owning bulk invoice voids either through the ClaimCenter user interface or by using a call to the BulkInvoiceAPI method <code>voidBulkInvoice</code> .
	pendingstop	This transition occurs if the owning bulk invoice stops either through the ClaimCenter user interface or by using a call to the BulkInvoiceAPI method <code>stopBulkInvoice</code> .
	pendingtransfer	This transition occurs if the check associated with the bulk invoice item transfers from the ClaimCenter user interface.
inreview	draft	This transition occurs if the bulk invoice item changes in such a way that the owning bulk invoice no longer passes validation.
notvalid	draft	This transition occurs if the bulk invoice item changes such that the owning bulk invoice no longer passes validation or the bulk invoice resubmits while in <code>pendingbulkinvoiceitemvalidation</code> status.
pendingvoid	voided	This transition occurs if the <code>updateBulkInvoiceStatus</code> method gets a value of <code>Voided</code> to confirm a successful void of the owning bulk invoice.
	stopped	This transition occurs if the <code>updateBulkInvoiceStatus</code> method gets a value of <code>Stopped</code> to confirm a successful stop of the owning bulk invoice.
	submitted	This transition occurs if the <code>updateBulkInvoiceStatus</code> method gets a value of <code>Issued</code> or <code>Cleared</code> to indicate a failed void attempt for the owning bulk invoice.
pendingstop	stopped	This transition occurs if the <code>updateBulkInvoiceStatus</code> method gets a value of <code>Stopped</code> to confirm a successful stop of the owning bulk invoice.
	voided	This transition occurs if the <code>updateBulkInvoiceStatus</code> method gets a value of <code>Voided</code> to confirm a successful void of the owning bulk invoice.
	submitted	This transition occurs if the <code>updateBulkInvoiceStatus</code> method gets a value of <code>Issued</code> or <code>Cleared</code> to indicate a failed stop attempt for the owning bulk invoice.
pendingtransfer	transferred	This transition occurs if the associated check transitions from <code>pendingtransfer</code> to <code>transferred</code> status.

Original bulk invoice item status	Changed status	How the status change occurs
		This transition happens if ClaimCenter received an acknowledgment for the associated message for the check changing to the pendingtransfer status. This transition requires you to call <code>check.acknowledgeTransfer</code> in your message ACK code in your messaging plugins.
voided	submitted	This transition occurs if the <code>updateBulkInvoiceStatus</code> method gets a value of <code>issued</code> or <code>cleared</code> to indicate a failed void attempt for the owning bulk invoice.
stopped	submitted	This transition occurs if the <code>updateBulkInvoiceStatus</code> method gets a value of <code>issued</code> or <code>cleared</code> to indicate a failed stop attempt for the owning bulk invoice.
transferred	Not applicable	This is an ending status.

Bulk invoice batch processes

ClaimCenter has several bulk invoice batch processes: `bulkinvoicesworkflow`, `bulkinvoicesescalation`, and `BulkInvoiceWF`.

See also

- *Administration Guide*

Deduction plugins

ClaimCenter supports plugins for checks and for other deductions.

Deduction calculations for checks

You can customize the logic that generates a list of deductions from a new primary check by implementing the `IBackupWithholdingPlugin` in ClaimCenter. It provides an integration point for automatically creating deductions from a payment. The plugin implementation must determine whether the deduction type applies and, if so, to create any applicable deductions.

Your plugin implementation must implement the `getDeductions` method on the plugin interface. This method takes a `Check` entity. Your implementation must generate zero or more `Deduction` entities to submit with the primary payment. The method must return an array of zero or more `Deduction` entities.

The properties you must set on each `Deduction` entity are listed below.

- `Amount` – The deduction amount
- `DeductionType` – The type of deduction being applied. This is a value in the `DeductionType` typelist. The type determines how to account for the deduction in your external financial system. For example, withholding for taxes.

ClaimCenter automatically sets the value of the `Check` property to refer to the current check, so you can leave this property `null`.

The built-in implementation of this plugin only calls the backup withholding utility class `gw.util.BackupWithholdingCalculator` to do the work. You can view and edit this Gosu class if you want to understand or modify the behavior.

ClaimCenter calls this plugin before the last step in the new check wizard. Your plugin implementation must identify deductions to a check, similar to taking taxes or benefits out of an employee paycheck. The net amount of the check must be less than payments charged to the claim file.

The most important example of this is backup withholding, which causes taxes to be withheld from the amount of the payment. This plugin implements the default behavior and gives you an opportunity to modify as appropriate.

Handling other deductions

ClaimCenter also includes another plugin definition for a similar purpose as the backup withholding plugin. This plugin interface is called `IDeductionAdapter`. This plugin is similar to the `IBackupWithholding` plugin interface. Differences include the following behaviors and requirements.

- The `IDeductionAdapter` handles deductions in a generic way, rather than for backup withholding for checks.
- `IDeductionAdapter` requires use of special template to generate parameters into a large `String` data object. In contrast, the `IBackupWithholding` plugin takes a typesafe `Claim` object.

The default implementation of the `IDeductionAdapter` plugin only calls the registered version of the `IBackupWithholding` plugin interface. In turn, that class calls the backup withholding utility class `gw.util.BackupWithholdingCalculator` to do all of its work. You can view and edit this file in Studio.

The plugin takes some data about the new primary check from the `Deduction_Check.gs` plugin template with the `check` root object and returns a list of deductions. A deduction is a much simpler object to generate than a reserve or a check. The template just needs enough information about the check and the payees to determine whether a deduction is necessary, and if so for how much.

Your plugin implementation must implement the `getDeductions` method on the plugin interface. Your implementation must generate zero or more `Deduction` entities to submit with the primary payment. The method must return an array of zero or more `Deduction` entities.

The properties you must set on each `Deduction` entity are listed below.

- `Amount` – The deduction amount
- `DeductionType` – The type of deduction being applied. This is a value in the `DeductionType` typelist. The type determines how to account for the deduction in your external financial system. For example, withholding for taxes.

ClaimCenter automatically sets the value of the `Check` property to refer to the current check, so you can leave this property `null`.

The deduction object just requires an amount and a type, which is a value in the `DeductionType` typelist. The type determines how to account for the deduction in your external financial system. For example, withholding for taxes. ClaimCenter automatically sets the value of the `Check` property to refer to the current check, so you can leave this property `null`.

part 6

Claim and policy integrations

ClaimCenter integrates with external systems to track claims on policies. ClaimCenter uses integrations to send information to or receive information from the external systems. Examples of such information are details of a policy that ClaimCenter sends to a claim management system and information about claims that a claim management system sends to ClaimCenter.

ClaimCenter uses both web services and plugins for data transfer.

Claim and policy integration

This topic describes web services, plugin interfaces, and tools for communicating with policy administration systems such as Guidewire PolicyCenter.

Policy system notifications

ClaimCenter provides a general architecture for notifications from ClaimCenter to policy administration systems. The only notification type in the base configuration is to detect large losses.

General purpose notification system

The ClaimCenter architecture for policy notifications is flexible enough to support multiple types of policy system notifications.

The main aspects of the ClaimCenter notification architecture are described below.

- Pre-update rules that detect specific types of issues and raise messaging events
- Notification handler classes for each type of policy system notification. Each one of these notifications correspond to one messaging event name. There is exactly one built-in notification handler class, which detects large losses.
- Event messaging rules that delegate work to the notification handlers to create messages to submit to the messaging queue. To do this, the rules call the handler's `createMessage` method.
- A messaging transport that gets each message (asynchronously in a separate thread) and asks the appropriate handler to send the message.
- A policy system plugin interface that defines the contract between ClaimCenter and an actual policy administration system. ClaimCenter includes a built-in version of this plugin interface that connects to PolicyCenter web services.
- Event message rules to handle resync

Each notification handler for each type of policy system notification corresponds to one messaging event name.

ClaimCenter calls the handler at the following times.

- In Event Fired rules, to create a message for the messaging queue
- At message send time to perform any last-minute actions then call the plugin to send the message
- If a claim is resynced

Each of these times correspond to different methods in the notification handler class. Each notification handler can define its own behavior at each of these times. If you make new notification types, Guidewire recommends following the general pattern of the built-in large loss notification class.

In the default configuration, ClaimCenter includes one notification handler: `LargeLossPolicySystemNotification`. This handler delivers large loss notifications to the policy administration system.

It is important to note that the handler mostly represents the type of notification itself, not necessarily all the implementation details. For instance, the built-in large loss notification handler delegates the actual work of sending the large loss message to the currently-registered version of the plugin interface `IPolicySystemNotificationPlugin`. If you make new notification types, Guidewire recommends following this general pattern and put your code that actually contacts the external system in your `IPolicySystemNotificationPlugin` implementation.

Large loss notification implementation details

The following table summarizes the rule sets that ClaimCenter uses for large loss notifications.

Purpose	Rule Set	Description
Pre-update rules to detect large losses	Large Loss Notification	<p>The <code>Preupdate > TransactionSetPreupdate > Large Loss Notification</code> rule is disabled by default in the base configuration. When enabled, the rule fires whenever transaction sets are created or changed. The rule's conditions determine if the claim exceeds the large loss threshold for the claim's policy type. They also determine if a message is in the queue or if PolicyCenter has been notified. If the conditions pass, the rule adds a <code>ClaimExceedsLargeLoss</code> event to the claim. You define the thresholds in the administration user interface of ClaimCenter.</p>
Event rules to support policy system notifications in general	Policy System Notification	<p>The <code>EventMessage > EventFired > Policy System Notification</code> rule is a general rule for all policy system notification events. In the base configuration, this rule is disabled. If enabled, the rules can do the following operations.</p> <ol style="list-style-type: none"> Determine the event name. For a large loss, the event name is <code>ClaimExceedsLargeLoss</code>. If you register additional notification handlers, the event name is different. Find the policy system notification handler that supports the current event name. For large loss, the handler class is <code>gw.policy.notification.LargeLossPolicySystemNotification</code>. The rules call the handler's <code>createMessage</code> method to create an outgoing message to persist to the database in the same transaction as the change that triggers the large loss notification. This call does not actually send the message. Sending the message happens asynchronously in another thread. <p>The handler's <code>createMessage</code> method delegates the actual sending behavior to the class that implements the <code>IPolicySystemNotificationPlugin</code> plugin interface.</p> <p>The <code>message.EventName</code> property encodes which notification occurred. The body of the message contains information that the notification handler might need. For large loss, for example, it would contain the size of the loss.</p> <p>At message send time, the messaging transport calls the notification handler's <code>send</code> method and does not use this rule set.</p> <p>If a <code>ClaimResync</code> claim resynchronization event occurs, ClaimCenter drops all queued (pending/errant) messages for the destination. However, ClaimCenter first calls the <code>EventFired</code> rule set to preserve and queue messages. These event rules trigger code that checks the notification handler's <code>MessageResyncBehavior</code> property for how to handle it. If the value is <code>COPY_LAST</code>, then only one pending message corresponding to that notification will be copied, the last one by send order.</p>

Notifying a policy administration system of a large loss

ClaimCenter can notify a policy administration system if a claim reaches a critical threshold, defined by policy type. The policy administration system can take appropriate actions to notify the policy's underwriter. Guidewire ClaimCenter is pre-configured to support this type of notification. When the claim exceeds the threshold, ClaimCenter creates a message using the messaging system. A special message transport plugin sends this message to your policy administration system and sends the claim's policy number, loss date, and the total gross incurred. This feature is called "large loss notification."

Guidewire PolicyCenter includes a built-in integration to support this notification from ClaimCenter. PolicyCenter publishes the web service `ClaimToPolicySystemNotificationAPI`. ClaimCenter calls this PolicyCenter web service across the network to add a referral reason and an activity in PolicyCenter.

If you use the built-in integration to PolicyCenter, you do not need to know the API details of the web service. For API details of this PolicyCenter web service, refer to the *PolicyCenter Integration Guide* in the PolicyCenter documentation set.

If you want to use a policy administration system other than PolicyCenter, write your own implementation of the plugin interface `IPolicySystemNotificationPlugin`. That plugin is the code that actually notifies the external system.

The definitions of the thresholds for what counts as a large loss are defined from within ClaimCenter, not in the policy administration system.

There is a `Claim` entity property called `LargeLossNotification` that tracks whether ClaimCenter has sent a notification for this claim. That property contains a typecode from the typelist `LargeLossNotification`.

PolicyCenter implementation of the large loss notification web service

The PolicyCenter implementation of the notification web service performs the following operations.

1. Finds the policy – Finds the policy from its policy number, and throws an exception if it cannot find it
2. Adds a referral reason – Creates a referral reason on the policy for the large loss
3. Adds an activity – Adds a new activity to examine the large loss.

On the PolicyCenter side, you can modify the built in implementation of this API to do additional things. You could also have different code paths depending on the size of the gross total of the loss.

Enabling large loss notification

To use the built-in plugin that connects to PolicyCenter, remember to set up your web services to connect to PolicyCenter. In Studio, in its Web Services editor, you can set the server, port, and authentication settings in that dialog.

If you want to define your authentication in Studio, set the web service to use HTTP authentication and specify your credentials there.

Alternatively, you can choose to leave the authentication setting on None and modify the code in the class `PCPolicySystemNotificationPlugin` to add authentication credentials. Look for the following code statement.

```
_policySystemAPI = new ClaimToPolicySystemNotificationAPI()
```

Immediately afterward, add a statement like the following and pass your user name and password.

```
_policySystemAPI.addHandler(new GAuthentificationHandler( "su", "gw" ))
```

See also

- *Installation Guide*

Using a policy administration system other than PolicyCenter

If you use a policy administration system other than PolicyCenter, you must write your own implementation of the `IPolicySystemNotificationPlugin` plugin interface. You send the notification to your policy administration system through whatever API is appropriate, which might be web services or another remote procedure call.

The remote procedure call to the external system must be synchronous because a messaging transport calls the plugin implementation as part of its send process. Assuming that your code throws no exceptions, when your plugin methods complete, the message transport acknowledges the message immediately afterward. The message transport must be certain the external system got the request before acknowledging the message.

The `IPolicySystemNotificationPlugin` plugin has only a single method that you must implement. The method is called `claimExceedsLargeLossThreshold` and takes the following arguments:

- A loss date as a `java.util.Date` object.
- A policy number as a `String`.
- The gross total incurred as a `String`, which must be convertible to a `gw.api.financials.CurrencyAmount` object. For example, specify 100 U.S. Dollars as "100 USD".
- A transaction ID as a `String`.

The transaction ID is an identifier that ClaimCenter creates. Use this ID to avoid processing the same notification twice. For example, suppose that ClaimCenter sends a notification and the remote system processes it, but the reply back to ClaimCenter never arrives due to network failure. ClaimCenter does not know that the notification was successfully delivered and retries the notification by using the same transaction ID. The remote system must detect that it has already processed a notification with this transaction ID and throw the exception `PolicySystemAlreadyExecutedException`. ClaimCenter catches this exception and marks the message as successfully delivered.

The plugin method in this interface can throw `PolicySystemRetryableException` or `PolicySystemAlreadyExecutedException`. Typically the plugin talks to a remote system with web services over the SOAP protocol. The exception class `PolicySystemRetryableException` indicates a temporary problem contacting the remote system. The exception class `PolicySystemAlreadyExecutedException` indicates that the remote system already processed a notification with that ID.

If you implement this plugin by using a web service, you must wrap any exceptions thrown by the web service. The web service throws exceptions specific to its WSDL, so you have to do processing similar to the following code block.

```
try {
    // call web service ...
} catch (e : ...web service exception...) {
    throw new gw.plugin.policy.PolicySystemRetryableException(e.message)
}
```

The message transport acknowledges the message

If no exceptions occur during the plugin notification method call, then, after the method completes, ClaimCenter marks the message to indicate that the policy administration system correctly received the message. This process is called message acknowledgment.

Additionally, each notification handler can set additional messaging-specific properties on the other objects as appropriate. For example, the large loss notification handler provided in the base configuration sets the claim property `LargeLossNotificationStatus`.

Add other notification types

To notify a policy administration system about a custom condition, you add a new notification type to ClaimCenter by extending `gw.policy.notification.PolicySystemNotificationBase`.

About this task

The only built-in notification type is the large loss notification, but you can add custom types.

Procedure

1. Create a new notification class that extends the class `gw.policy.notification.PolicySystemNotificationBase`. That base class contains some default behavior.
2. In your class definition, set up the ClaimCenter event name that corresponds to your notification handler. Also set up a static method to instantiate your class. Follow the approach in the large loss handler, as shown in the following lines of code.

```
/** The event name for this notification */
public static final var EVENT_NAME : String = "ClaimExceedsLargeLoss"
```

```
/** Singleton instance of this notification strategy */
public static final var INSTANCE : LargeLossPolicySystemNotification =
    new LargeLossPolicySystemNotification()

private construct() {
    super(EVENT_NAME)
}
```

Replace `LargeLossPolicySystemNotification` with your class name and replace the `String` value `"ClaimExceedsLargeLoss"` with your event name.

3. Implement the `createMessage` method to generate a message. This method takes a message context object, which is the object that Event Fired rules get to assist in message creation. The large loss notification uses code like that shown below.

```
override function createMessage(context : MessageContext) {
    var claim = context.Root as Claim
    var amt = FinancialsCalculationUtil.getTotalIncurredGross().getAmount(claim)
    var renderedAmt = CurrencyUtil.renderAsCurrency(amt)
    var msg = context.createMessage(renderedAmt)
    msg.MessageRoot = claim
    claim.LargeLossNotificationStatus = "InQueue"
}
```

Note that it sets a field on the claim itself to indicate the claim notification is in queue. Also note that the method does not need to indicate what type of notification it is. Even if you add additional notifications, you do not need to add a special property to the message entity to identify the type of notification. Instead, your message code that runs later can detect which notification type by checking the `message.EventName` property.

4. Implement the `send` method, which ClaimCenter calls from the messaging transport that gets messages from the send queue. The message transport gets messages one by one from the send queue in another thread. Any changes on the message happen in different database transactions than the original change that triggered creation of the message. The `send` method takes a reference to the instance of the policy system notification plugin, the `Message` entity, and the transformed payload `String`. The transformed payload is a variant of `Message.Payload` and is different if the destination used a `MessageRequest` or `MessageBeforeSend` plugin in addition to the `MessageTransport` plugin. Generally speaking, use the transformed payload rather than `Message.Payload`. Your version of this method might just do what large loss does, which is call the plugin method that matches your notification type. If you added additional methods to your plugin implementation, you will need to cast the plugin reference to your specific implementation class type before calling custom methods on it. The large loss version looks like the code statements shown below.

```
override function send(plugin : IPolicySystemNotificationPlugin, message : Message,
                      transformedPayload : String) {
    var claim = message.Claim
    plugin.claimExceedsLargeLossThreshold(claim.LossDate, claim.Policy.PolicyNumber,
                                          transformedPayload, message.PublicID)
}
```

5. Implement the `afterSend` method. ClaimCenter calls this immediately after sending the message. You can use this as a hook to set messaging-specific fields on the `Claim`.

```
override function afterSend(message : Message, status : MessageStatus) {
    if (status == GOOD) {
        message.Claim.LargeLossNotificationStatus = "Sent"
    } else if (status == NON_RETRYABLE_ERROR) {
        message.Claim.LargeLossNotificationStatus = "None"
    }
}
```

6. Implement the `MessageResyncBehavior` property getter. ClaimCenter responds differently during resync based on the value.

- If the value is `DROP`, ClaimCenter does not resend the notification during resync.
- If the value is `COPY_LAST`, then ClaimCenter copies only one pending notification message corresponding to that notification type for that messaging destination for that claim. ClaimCenter uses the last one, by send order.

- If the value is COPY_ALL, then ClaimCenter copies all pending notification messages for that notification type for that messaging destination for that claim.

The following code demonstrates a simple implementation.

```
override property get MessageResyncBehavior() : MessageResyncBehavior {
    return COPY_LAST
}
```

7. Register your notification type handler with the system by modifying the class `gw.policy.notificationPolicySystemNotificationList`. This class is a configurable list of all policy system notifications. Adding a notification to this list makes the notification known to the policy system notification event messaging rules and to the messaging transport `PolicySystemNotificationMessageTransport`. Although you cannot add new methods to the actual interface for the `IPolicySystemNotificationPlugin`, you can add new notifications to this list. If you add notifications, override the notification send method to directly call the mechanism that calls the policy administration system.

The following code shows the built-in version of this list.

```
class PolicySystemNotificationList {
    /** List of all available notifications */
    public static var ALL : List<PolicySystemNotificationBase>
        = { LargeLossPolicySystemNotification.INSTANCE }.freeze()

    /** Never instantiated */
    construct() {}
}
```

To add another notification class called `abc.integration.MyNotificationHandler`, change the list to look like the following code block.

```
class PolicySystemNotificationList {
    /** List of all available notifications */
    public static var ALL : List<PolicySystemNotificationBase>
        = { LargeLossPolicySystemNotification.INSTANCE ,
            abc.integration.MyNotificationHandler.INSTANCE }.freeze()

    /** Never instantiated */
    construct() {}
}
```

8. Add new preupdate rules that detect the new condition. If you detect the condition, raise the event with code such as the following statement.

```
claim.addEvent("MyEventName")
```

9. If you have not already done so, enable the notification system rules and plugins.
10. Test your notification system with a development version of ClaimCenter and your policy administration system.

Policy search plugin

The policy search plugin enables ClaimCenter users to search for a policy, review the list of possible choices, select a specific policy, and retrieve the full details of that policy. The policy details are stored with the claim as a local copy. The `IPolicySearchAdapter` plugin interface defines a search method and a retrieve method to support standard policy search and retrieval. This plugin interface defines an additional retrieval method to handle the special case of policy refresh, which is discussed later in this topic.

If you use Guidewire PolicyCenter, you can use a built-in implementation of this plugin that queries PolicyCenter for policies. To use this plugin, in Guidewire Studio™, navigate in the **Project** window to **configuration > config > Plugins**, and then open `IPolicySearchAdapter.gwp`. Then register the Gosu class `gw.plugin.pcintegration.pc1000.PolicySearchPCPlugin`.

If you use a policy administration system other than PolicyCenter, you must write your own implementation of this plugin interface.

The base configuration of ClaimCenter also includes a demonstration version of this plugin implementation, which generates example policies.

To understand the flow of ClaimCenter policy search, you must distinguish the two concepts of search and retrieve from the context of ClaimCenter.

Search means getting policy summaries, possibly many of them.

A search operation involves a call to an external system to retrieve policy summaries only and returning zero, one, two, or a very large number of results. The `searchPolicies` method receives a set of criteria and returns a subset of the full policy information, which in the typical case determines the policy that the user wants. The return result is a `PolicySearchResultSet`. This object has a `Summaries` property that contains an array of `PolicySummary` entities—an array of policy summaries.

Retrieval means getting a single, unique, full policy.

The two `retrieve` methods take policy summaries that contain a specific policy number, an effective date, such as a loss date, and other search information. They retrieve a single, unique version of the policy that is effective as of a specific effective date.

See also

- *Installation Guide*

Multicurrency and policies

If you use multicurrency mode, you must set the currency on the policy in the `Policy.Currency` property.

Typical chronological flow of policy search and retrieval

The chronological flow for policy search and retrieval is described below. In particular, note the distinction between the words search and retrieve, and between summaries and entire policy.

1. The user requests a policy search in the ClaimCenter user interface.
2. The policy search user interface pages use a PCF page to display the search fields.
3. The PCF page encodes user-entered data in properties in a `PolicySearchCriteria` object. This object is non-persistent—it is described in the ClaimCenter Data Dictionary, but has no corresponding table in the database for persisting it.
4. To begin the search, ClaimCenter calls `IPolicySearchAdapter.searchPolicies(PolicySearchCriteria)`, which returns a `PolicySearchResultSet` entity. This entity contains an array of `PolicySummary` entities to display as summaries for the user to select.

At some future time, ClaimCenter must retrieve the entire policy, not merely the summary. At that time, ClaimCenter uses the `PolicySummary` entity to retrieve the policy. Because of this usage, the `PolicySummary` must contain all the information required for policy retrieval.

If any policy-related properties are required to retrieve a single unique policy, store that information in the `PolicySummary` entity. Extend the data model with new properties as necessary to store that information. For example, suppose at policy retrieval time you want to return only a single vehicle on the policy based on special search information passed to the search. Copy that information from the `PolicySearchCriteria` to each `PolicySummary` entity.

5. The selected summary is stored in the `NewClaimPolicyDescription` entity on the `NewClaimPolicy` object, which is part of the wizard user interface.
6. The user selects a summary to be retrieved by using the user interface.
7. To retrieve a single unique policy from the external system, ClaimCenter calls `IPolicySearchAdapter.retrievePolicyFromPolicySummary(PolicySummary)`. This method must return a `PolicyRetrievalResultSet`, which contains a `Policy` entity.
8. If you refresh the `Policy` in the user interface, ClaimCenter calls the `IPolicySearchAdapter.retrievePolicyFromPolicy` method with the current local version of the policy. If

you need any properties from the original policy search or retrieval at the time of policy refresh, you must set them on the **Policy** entity during original policy retrieval. The `retrievePolicyFromPolicy` method can use these properties. This method also returns a `PolicyRetrievalResultSet` entity.

Implement the plugin method `retrievePolicyFromPolicy` to support ClaimCenter refreshing the policy.

In the typical case of searching for a policy, the following methods are called.

- Clicking the **Search** button calls the `IPolicySearchAdapter` method `searchPolicies`.
- Clicking the **Next** button to leave the first page calls the `IPolicySearchAdapter` method `retrievePolicyFromPolicySummary`.

If you never return to the first page, there are no more calls to the plugin's `searchPolicies` method.

The path is slightly different for commercial auto and commercial property policies, for which there is an extra page in which you can select risk units. That page is titled either **Select Involved Policy Vehicles** or **Select Involved Policy Properties**. For these types of policies, the `retrievePolicyFromPolicySummary` method call happens only after you click the **Next** button on that extra risk unit selection page.

It does not matter what kind of search criteria you use, such as policy number or policy type. ClaimCenter calls `searchPolicies` once and then calls `retrievePolicyFromPolicySummary`.

Every time you use the **Search** button, ClaimCenter calls the plugin's `searchPolicies` method. For example, if you do 10 searches, ClaimCenter calls that method 10 times.

If you do not search for a policy, you are creating a new, unverified policy. This approach does not trigger a call to the `IPolicySearchAdapter` at all. However, you can go back to the first page and click the **Search** button to change your approach and search for verified policies.

ClaimCenter calls `searchPolicies` once per search, and calls `retrievePolicyFromPolicySummary` again when you click **Next**.

Policy search example and additional information

The following example shows a simple policy search plugin's main function in Java.

```
public PolicySearchResultSet searchPolicies(PolicySearchCriteria policySearchCriteria)
    throws RemoteException {
    int maxPolicySearchResults = 25; // must agree with MaxPolicySearchResults in config.xml
    // Get whatever properties you need from the PolicySearchCriteria...
    LossType lossType = policySearchCriteria.getLossType();
    String vin = policySearchCriteria.getVin();
    PolicyType policyType = policySearchCriteria.getPolicyType();
    String propertyCity = policySearchCriteria.getPropertyAddress().getCity();

    // [At this point, send these properties across to a remote system and get results back]
    // Populate the result object with an array of policy summaries
    // In this case just return one result
    PolicySearchResultSet policySet =
        (PolicySearchResultSet)EntityFactory.getInstance().newEntity(PolicySearchResultSet.class);
    PolicySummary policySummary =
        (PolicySummary) EntityFactory.getInstance().newEntity(PolicySummary.class);
    policySummary.setPolicyNumber("00-00001");

    // [Set any other required properties...]
    PolicySummary[] policySummaryArray = new PolicySummary[1];
    policySummaryArray[0] = policySummary;
    policySet.setSummaries(policySummaryArray);
    policySet.setUncappedResultCount(1);

    if (policySet.getUncappedResultCount() > maxPolicySearchResults) {
        policySet.setResultsCapped(true);
    } else {
        policySet.setResultsCapped(false);
    }

    return policySet;
}
```

If your code path does not support a very large number of results, you can set a maximum number of results to show. You set this number in the `MaxPolicySearchResults` configuration parameter in the `config.xml` file. In the default

configuration, this parameter is set to 25. This setting affects only the number of rows that ClaimCenter shows. The plugin must observe the same limitation during search. Never return more than that maximum number of results.

Set the `PolicySearchResultSet.UncappedResultCount` property to the number of results found according to the search criteria, even if this number exceeds the number of results actually returned in the result set. Also set the `PolicySearchResultSet.ResultsCapped` property to `true`. If you do so, the ClaimCenter user interface can show a message that informs the user when there are “too many results found to display.”

You can also add as many additional search criteria for policy searches as you want by performing the following operations.

1. Extend the `PolicySearchCriteria` object’s data model with new extension properties just like you would extend the data model of any other ClaimCenter entity.
2. If appropriate, modify the PCF pages to prompt for new user data that relate to those properties. Alternatively, the new search criteria could be set programmatically based on some other factors other than new user data. For instance, the context within the ClaimCenter user interface could be such a factor.
3. Alter the PCF files appropriately to set these extension properties from form field data.

Sometimes policies have a very large number of insured risks, such as a large number of insured vehicles or buildings. In some cases, you might need details only for a specific risk involved in the claim. For example, the search criteria can identify the specific risk by passing a vehicle’s Vehicle Identification Number (VIN). By adding this search criterion, the policy search plugin can return only the coverages for that risk with the policy summary. If the plugin does not use some search criteria, ignore that search criteria data. It might be appropriate to remove those search fields from the user interface if they are always ignored.

Policy retrieval additional information

Policy retrieval involves returning a single policy, rather than the list of matches returned by a search.

The return result of the `retrievePolicyFromPolicySummary` plugin method is an entity called `PolicyRetrievalResultSet`. It has properties of its own but is mostly a shell for a `Policy` entity within the `PolicyRetrievalResultSet.Result` property.

After this method retrieves the full details for a policy, it creates a new `Policy` object and populates its properties from data from the external system. The method adds that data to a new `PolicyRetrievalResultSet` entity and returns it.

ClaimCenter creates a `Policy` object from a `CCPolicy` object retrieved from PolicyCenter. As part of this process, ClaimCenter copies fields on the `CCPolicy` object that have matching `Policy` field names and data types into the `Policy` object. For any fields that have different names or fields that have different data types, ClaimCenter uses the `pc-to-cc-data-mapping.xml` file to see if the fields are defined there.

- For fields with different names, an XML entry can designate the name that ClaimCenter uses in the `Policy` object.
- For fields with different data types, an XML entry can designate a mapping class to use in mapping the data types.
- If a `CCPolicy` field does not exist in `Policy` and is not defined in `pc-to-cc-data-mapping.xml`, ClaimCenter ignores that field when creating the `Policy` object.

For example, in the base configuration, the `CCPolicy.Account` field must be renamed in `Policy` to `AccountNumber`, and its data type must be transformed. The XML entry in `pc-to-cc-data-mapping.xml` defines both actions and uses the class `NameTranslatingFieldMapper` to do the data transformation. In `pc-to-cc-data-mapping.xml`, the XML code is shown below.

```
<EntityMapping source="CCPolicy" target="Policy">
  <FieldMapping source="Account"
    mapperClassName="gw.plugin.integration.mapping.NameTranslatingFieldMapper">
    <MapperProperty name="NewFieldName" value="AccountNumber"/>
  </FieldMapping>
  ...

```

If the `PolicyCenter` object has a name that does not match the pattern `object-name-on-ClaimCenter`, you can define an entry in `pc-to-cc-data-mapping.xml` to handle this difference.

The Policy object graph is fairly complicated. All objects in the graph must be populated correctly. One aspect that is particularly difficult is associating contacts, both people and companies, as the insured, the agent, an interested party, and so on. These associations are set by linking the contact to the policy and describing the ways in which the contact is related as a set of roles.

The definition for these subobjects and roles is in `pc-to-cc-data-mapping.xml`. These XML entries specify the `ContactArrayFieldMapper` class to do the actual work of populating `Policy` with the subobjects. For example, for the `CoveredParty` role on a `Policy`, the mapping is defined as shown below.

```
<EntityMapping source="CCPolicy" target="Policy">
...
  <FieldMapping source="CoveredParty"
    mapperClassName="gw.plugin.pcintegration.pc1000.mapping.ContactArrayFieldMapper"/>
...

```

Refer to the *Data Dictionary* for the complete set of properties on the `Policy` entity.

There are three possible outcomes of a policy retrieval, and the cases can be distinguished by values in the `PolicyRetrievalResultSet`.

Successful

The retrieval parameters map to a single, unique policy. The result entity has a `Result` property that contains a `Policy` entity, rather than `null`. Also, the result entity has a `NotUnique` property set to `false`.

Unsuccessful, multiple matches

Unsuccessful because retrieval parameters map to multiple policies. The result entity has a `Result` property that contains a `Policy` entity, rather than `null`. The result entity has a `NotUnique` property set to `true`.

Unsuccessful, no match

Unsuccessful, because retrieval parameters do not map to any policies. The result entity has a `Result` property that contains `null`.

One way to implement policy retrieval is to return the entire policy, including all subobjects. However, you can choose to save bandwidth and return only a subset of the vehicles or real estate properties. Perhaps return only objects included in the original search criteria that triggered the search, the summary of which a user clicked on, and thus triggered a retrieval. For example, if a search was triggered based on a specific vehicle, that information can be used only to retrieve that vehicle and not other vehicles on the policy. If you want to do this, be careful to note the following special properties on the `Policy` entity.

- If a policy insures vehicles, your plugin can return a subset of vehicles, perhaps only one vehicle. If you return a subset of vehicles, set the `Policy.TotalVehicles` property to the actual number of insured vehicles. Do not set it to the number of vehicles in `Policy.Vehicles`. Setting the total correctly allows the user interface to notice this difference between the two numbers. The application shows a message saying that not all vehicles on the policy are available in ClaimCenter.
- If a policy insures real estate properties, your plugin can return a subset of real estate properties, perhaps only one real estate property. If you return a subset of properties, set the `Policy.TotalProperties` property to the actual number of insured real estate properties. Do not set it to the number of real estate properties in `Policy.Properties`. Setting the total correctly causes the user interface to show the difference between the two numbers. The application shows a message saying that not all real estate properties on the policy are available in ClaimCenter.

The following example shows what to do in the policy retrieval plugin implemented in Gosu.

```
override function retrievePolicyFromPolicySummary( policySummary : PolicySummary ) :
  PolicyRetrievalResultSet

  var p = new Policy()
  p.policyNumber = "123"
  p.policyType = PolicyType.TC_AUTO_PER
  p.policyStatus = PolicyStatus.TC_EXPIRED
  p.isVerified = true

  var insured = new Person();
  insured.EmailAddress1 = "insured@testmail.com"
  insured.EmailAddress2 = "jdoe@central.org"
```

```

insured.FirstName = "insured_firstname"
insured.HomePhone = "532-453-0989"
insured.LastName = "insured_lastname"
insured.LicenseNumber = "CL50976800"
insured.LicenseState = State.TC_CA
insured.Occupation = "Technical Writer"
insured.Preferred = Boolean.TRUE
insured.Prefix = NamePrefix.TC_MR
insured.PrimaryPhone = PrimaryPhoneType.TC_WORK;

// Set all your other properties on the Policy entity.
...

var policySet = new PolicyRetrievalResultSet();
policySet.NotUnique = Boolean.FALSE // if successful, and there is a single match
policySet.Result = p

// Return the result set.
return policySet
}

```

Modifying the New Claim wizard to reload a policy

The New Claim wizard retrieves policy information from an external system. On the first page of the wizard, the user can search for a policy.

ClaimCenter searches for a policy in the external policy system or creates a new policy, based on the current policy description. If the policy description matches the claim's current policy, the wizard's `setPolicy` method does nothing.

A policy search uses the registered policy search plugin and returns a list of policy summaries, which are simplified versions of policies. ClaimCenter shows these summaries in a list view so the user can choose a policy. When the user chooses a policy and clicks **Next**, the wizard calls the policy search plugin to retrieve the policy for the picked summary. ClaimCenter sets this policy as the claim's policy.

However, suppose at this point that the user returns to the first step in the wizard, picks a new policy, and clicks **Next** again. ClaimCenter must determine whether to keep and reuse the current policy or request that the policy search plugin retrieve another policy.

ClaimCenter makes this decision by calling the method `isSamePolicy` on a policy summary, which compares the currently picked summary with the policy that is on the claim.

- If `isSamePolicy` returns `true`, the wizard keeps the claim's current policy.
- If `isSamePolicy` returns `false`, the wizard replaces the claim's current policy with a new one, fetched with the policy search plugin.
- If the user explicitly picks a different external policy, `isSamePolicy` returns `false`, which causes ClaimCenter to refresh the policy.

To decide whether two policies are the same, `isSamePolicy` compares a set of fields to see if they match. By default, the policy summary fields that this method checks are `PolicyType`, `PolicyNumber`, `EffectiveDate`, and `ExpirationDate`. If you want the method to compare additional fields, modify the enhancement file `PolicySummary.gsx`, which implements the policy summary `isSamePolicy` method. You can add code to customize how ClaimCenter compares policy summaries to determine if two policies match.

For example, suppose you want two policy summaries with different loss dates to trigger refreshing the policy. Open `PolicySummary.gsx` and find the method `isSamePolicy`. The method has the following for loop:

```

for (policySummaryProperty in properties) {
    if (this.getFieldValue(policySummaryProperty) !=
        policy.getFieldValue( policySummaryProperty )) {
        return false;
    }
}

```

After the `for` loop, add the following code:

```

//Test to see if the LossDate is different
if ( this.getFieldValue("LossDate") !=
    policy.Claim.getFieldValue("LossDate") ){
    return false;
}

```

Note: Use this pattern for any policy properties that you want to test as part of the policy summary.

Claim search web service for policy system integration

If you use a supported version of PolicyCenter, PolicyCenter uses the ClaimCenter web service `PCClaimSearchIntegrationAPI` web service to retrieve claim summaries based on search criteria. Typically you do not need to call this web service directly. PolicyCenter calls this web service automatically if you configure PolicyCenter to connect to ClaimCenter for claim search.

If you use a policy administration system other than PolicyCenter, your policy administration system can call this web service. However, this web service is written specifically to support a data model very similar to what PolicyCenter needs. If you use a policy administration system other than PolicyCenter, consider writing your own custom web services that directly address integration points between ClaimCenter and your specific policy administration system.

The `ClaimAPI` contains some similar methods. For example, the `ClaimAPI` web service provides the `findPublicIdByClaimNumber` method and the `PCPolicySearchIntegrationAPI` web service provides the `getClaimByClaimNumber` method. However, neither web service returns a complex claim graph. Instead, add custom web services to search for claims or export claim summaries in the data model format your policy administration system needs. Only export the claim fields and subobjects that are necessary.

Search for claims

If PolicyCenter or another policy administration system wants to identify a claim by a claim number and an effective date, it can call the `searchForClaims` method in this interface. It takes a `PCClaimSearchCriteria` object, which encapsulates the search criteria from the policy administration system. It includes the properties described in the following table.

Property	Type	Description
<code>BeginDate</code>	<code>Calendar</code>	The begin date
<code>EndDate</code>	<code>Calendar</code>	The end date
<code>Lob</code>	<code>String</code>	Line of business, specified by its name
<code>PolicyNumbers</code>	<code>String[]</code>	An array of policy numbers, as an array of <code>String</code> values

Set some or all of these properties and pass the `PCClaimSearchCriteria` to the `searchForClaims` method.

The result is an array of `PCClaim` entities, which are effectively like claim summary objects. They do not contain the full claim detail or object graph. Each object contains the following fields.

Property	Type	Description
<code>ClaimNumber</code>	<code>String</code>	The claim number
<code>LossDate</code>	<code>Calendar</code>	The loss date
<code>PolicyNumber</code>	<code>String</code>	The policy number of the best matched policy
<code>PolicyTypeName</code>	<code>String</code>	The policy type name
<code>Status</code>	<code>String</code>	The claim status, as a string
<code>TotalIncurred</code>	<code>BigDecimal</code>	The total incurred amount on the claim.

The following example uses the `PCClaimSearchCriteria` constructor to assemble the object, and then calls the web service method.

```
String[] polNums = { "abc:234234", "abc:298734" };
PCClaimSearchCriteria crit = new PCClaimSearchCriteria(beginDate, endDate, theLOBName, polNums);
// Call the API
```

```
PCClaim results = pcclaimSearchCriteria.searchForClaims(crit);  
// Get results  
print(results[0].ClaimNumber);  
print(results[0].Status);  
print(results[1].ClaimNumber);  
print(results[1].Status);
```

Get number of matched claims

If you want to get only the number of results that match, PolicyCenter or another policy administration system calls the `getNumberOfClaims` method. It takes the same `PCClaimSearchCriteria` as the `searchForClaims` method. The `getNumberOfClaims` method returns an integer number of matching claims.

The following example uses the `PCClaimSearchCriteria` constructor to assemble the object, and then calls the web service method.

```
String[] polNums = { "abc:234234","abc:298734" };  
PCClaimSearchCriteria crit = new PCClaimSearchCriteria(beginDate, endDate, theLOBName, polNums);  
  
// call the API  
int totalResults = pcclaimSearchCriteria.searchForClaims(crit);
```

Get claim detail

If PolicyCenter or another policy administration system wants more detail, it calls the `PCClaimSearchIntegrationAPI` web service method `getClaimByClaimNumber`. This method takes a claim number and returns a `PCClaimDetail` entity. This object contains much more information about a claim, such as the remaining reserves and the loss cause.

User claim view permission

PolicyCenter can directly direct a PolicyCenter user to the ClaimCenter user interface. To make this process work smoothly, it might be necessary to give a PolicyCenter user permission to view a claim. PolicyCenter calls the ClaimCenter `PCPolicySearchIntegrationAPI` web service method `giveUserClaimViewPermission` to give the user permission to view the claim. This method takes a claim public ID and a user name, both as `String` values.

ClaimCenter exit points to PolicyCenter and ContactManager

There are screens in the ClaimCenter user interface in which a user can directly open another application in a separate browser window. This feature is implemented by using the PCF widget `ExitPoint`, which in the base configuration sends a URL to a corresponding PCF widget called `EntryPoint` in the target Guidewire application.

In the base configuration, ClaimCenter provides `ExitPoint` widgets that can perform the following operations.

- Open PolicyCenter to directly view and edit policy information.
 - The file `ClaimPolicyGeneral.pcf` provides a **View Policy in Policy System** button that uses the `ViewPolicy` exit point.
 - The file `PolicyLocationSearchResultsLV.pcf` provides a cell in the search results list view that uses the `ViewPolicy` exit point.

To set the URL for the exit point to PolicyCenter, edit the configuration parameter `PolicySystemURL` in `config.xml`. Set this parameter to the URL for the server as shown in the sample URL below.

```
http://localhost:8180/pc
```

These exit point configuration parameters are configured in `config.xml` and not in `suite-config.xml`. The configuration parameters in `suite-config.xml` support integration between Guidewire applications through web services. The exit point configuration parameters in `config.xml` support integration between web browsers.

- Open ContactManager to directly view and edit contact information. The file `AddressBookSearchScreen.pcf` provides an **Open in ContactManager** button that uses the `GoToAB` exit point.

The URL for the exit point to ContactManager is defined in the ClaimCenter file `suite-config.xml`.

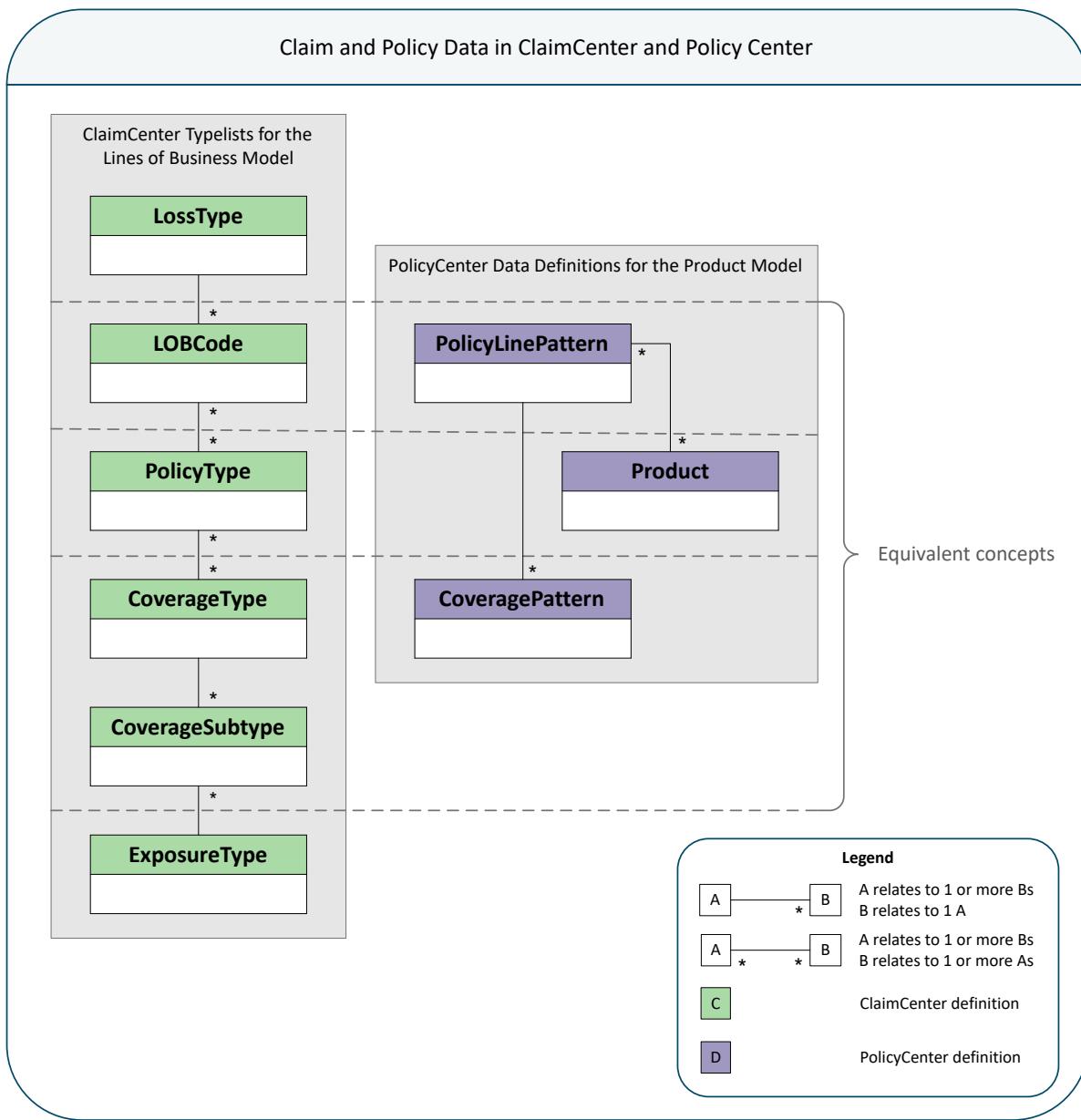
```
http://localhost:8280/ab
```

You can integrate with a system other than a Guidewire core application, such as a policy system that is not Guidewire PolicyCenter. If you want to send the system additional parameters, you can add them to the URL configuration parameter for that system. If the configuration parameter is absent or commented out or set to the empty string, exit point buttons for that system in the user interface are hidden.

PolicyCenter Product Model import into ClaimCenter

If you run instances of ClaimCenter and PolicyCenter together, you must keep your ClaimCenter lines of business model synchronized with your PolicyCenter product model. If you change your PolicyCenter product model, you must merge the changes with your ClaimCenter lines of business model to ensure synchronization. To help you synchronize your ClaimCenter lines of business model with your PolicyCenter product model, PolicyCenter provide the ClaimCenter Typelist Generator command-line tool.

Some typelists in the ClaimCenter lines of business model use data definitions from the PolicyCenter product model as the following diagram shows.



For example, LOB codes in ClaimCenter are equivalent to policy lines in PolicyCenter. Policy types are equivalent to products, and coverage types are equivalent to coverages. The generator adds new LOB typecodes to the ClaimCenter LOB typelist that correspond to codes for new PolicyCenter policy lines. The generator adds new typecodes to the typelists for policy type and coverage type in a similar way.

The code that you specify for a new product model pattern in the Product Designer becomes the `codeIdentifier` attribute of that element in the product model. The Product Designer generates a new public ID for the new product model pattern. The maximum length of a public ID is 64 characters. The maximum length of a code identifier for a policy line pattern or a product pattern is 64 characters. The maximum length of a code identifier for other product model patterns is 128 characters.

The generator uses the `public-id` attribute of the product model pattern and the `code` attribute of a ClaimCenter typecode in the equivalent typelist to match products and typecodes. If the `public-id` attribute of the product model pattern does not match the `code` attribute of any ClaimCenter typecode in the equivalent typelist, the generator creates a new typecode. The generator uses the `codeIdentifier` attribute of the product model element to add an `identifierCode` attribute to all typecodes that do not have a code identifier. The value that you use in a program to refer to the typecode is `TYPELIST.TC_IDENTIFIERCODE`.

The ClaimCenter lines of business model also has typelists in its hierarchy above and below the PolicyCenter equivalents. For example, LOB codes in ClaimCenter link to loss types. The generator does not link LOB codes to loss types. You must link new LOB codes to their parent loss types manually in Guidewire Studio for ClaimCenter. Similarly, at the bottom of the hierarchy, you must link new coverage types and coverage subtypes from PolicyCenter to exposure types in ClaimCenter.

See also

- *Gosu Reference Guide*

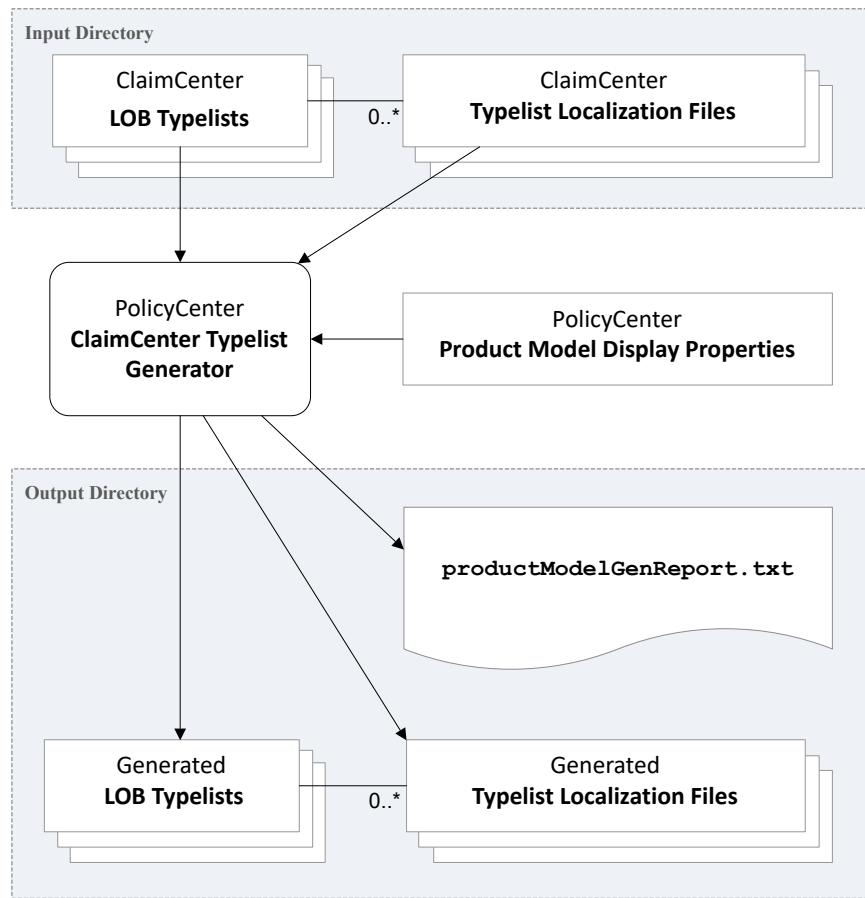
Configuring the ClaimCenter Typelist Generator

When you run the ClaimCenter Typelist Generator, you specify the following options.

Option	Description
Input directory	Location from where the generator reads ClaimCenter typelists and typelist localization files.
Output directory	Location from where the generator writes ClaimCenter typelists and typelist localization files.
Map new coverages to general damage exposure	Specify whether the generator associates new coverages from PolicyCenter with the generic General Damage exposure type in the base configuration of ClaimCenter. This option is particularly helpful during a configuration's initial-development phase.
ClaimCenter version	Specify the ClaimCenter version relevant to the input and output LOB typelists.

The following diagram illustrates the basic operation of the ClaimCenter Typelist Generator. The Typelist Generator processes files from the input directory. Using Product Model Display Properties in PolicyCenter, the Generator creates LOB typelists, typelist localization files, and a report (`productModelGenReport.txt`) in the output directory.

ClaimCenter Typelist Generation



ClaimCenter Typelist Generator input files

The ClaimCenter Typelist Generator reads the following ClaimCenter typelist files as input.

- CoverageSubtype.ttx
- CoverageType.ttx
- CovTermPattern.ttx
- ExposureType.ttx
- LOBCode.ttx
- LossPartyType.ttx
- PolicyType.ttx

Always put the latest versions of your ClaimCenter lines of business typelist files in the input directory. With input files, the generator preserves links between LOB codes and loss types that you configured in Guidewire Studio for ClaimCenter. In a similar way, providing input files preserves links between coverage types and exposure types. The generator also preserves typecodes that exist in ClaimCenter from other, third-party policy administration systems.

The ClaimCenter Typelist Generator reads in and writes out the **ExposureType.ttx** file only if you configure the generator to map new coverages to the General Damage exposure type.

Add typelists to the ClaimCenter Typelist Generator

About this task

You can add ClaimCenter typelists to the typelists that the ClaimCenter Typelist Generator processes.

Procedure

1. In PolicyCenter Studio, open `ProductModelTypelistGenerator.gs` in the `gsrc.gw.webservice.pc.pcxxxx.ccintegration` package, where `xxxx` is the release number. This value must be `1000` or greater.
2. In `_covTermPatternCCAddedTypelistCategories`, add the name of the typelist.

ClaimCenter Typelist Generator input and output directories

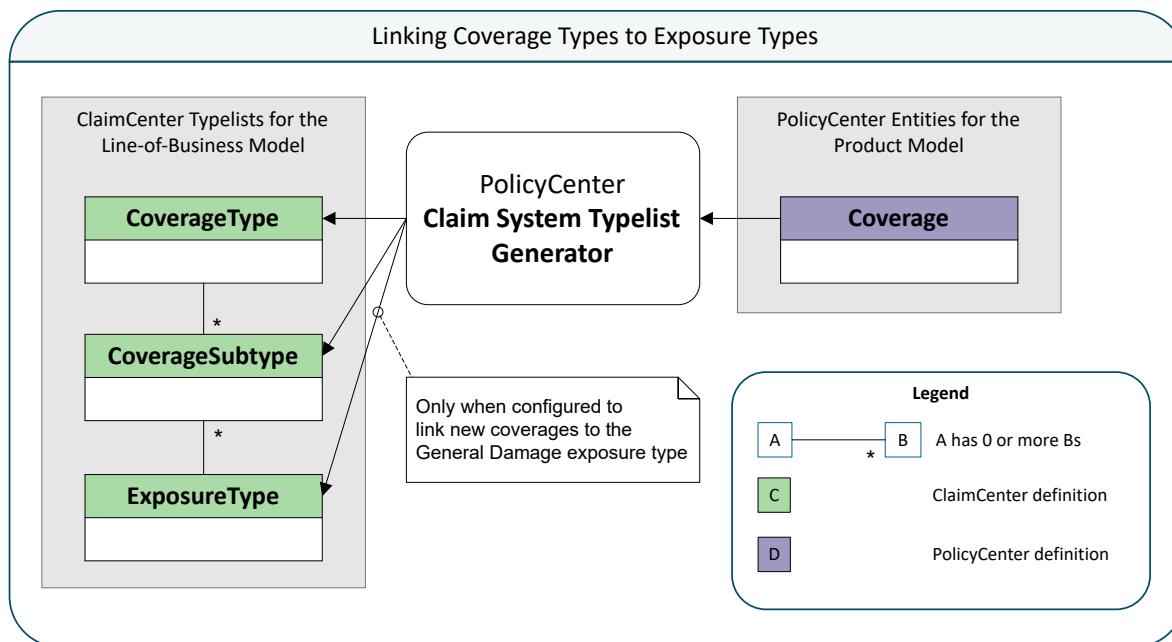
The ClaimCenter Typelist Generator can use the same directory for the input and output directories. The generator reads input files when it starts and writes output files when it finishes. When the input and output directories are the same, your typelist and properties files are overwritten with generated changes. This arrangement works well for demonstration situations. However, using the same directory for input and output prevents you from using a difference tool to determine the typecodes that the generator changed.

In a development or demonstration environment, you generally set up a PolicyCenter and a ClaimCenter instance on the same machine. If so, the generator can use the ClaimCenter directory that holds the lines of business typelist files as the input and output directories. This approach avoids steps to manually copy files from and to ClaimCenter.

In a production environment, do not configure the ClaimCenter Typelist Generator to use ClaimCenter directories for input or output.

Generated coverage subtypes

The ClaimCenter Typelist Generator takes Coverage entities in PolicyCenter and generates `CoverageType`, `CoverageSubtype`, and `ExposureType` typelists in ClaimCenter. In the ClaimCenter lines of business model, you link coverage types to exposure types through coverage subtypes, as shown in the following illustration.



Coverage subtypes duplicate their coverage types to implement many-to-many relationships between coverages and exposures on claims.

For new coverages in PolicyCenter, the generator creates corresponding coverage types and subtypes in `CoverageType.ttx` and in `CoverageSubtype.ttx`. To link generated coverage types to generated subtypes, the generator adds the types and subtypes as categories of each other. Use the generated subtypes to link corresponding coverage types to exposure types in Guidewire Studio for ClaimCenter.

The ClaimCenter Typelist Generator adds generic subtype typecodes in `CoverageSubtype.ttx` for new coverage types. The generic coverage subtypes have the same codes, names, and descriptions as the corresponding coverages. Typically, you link a coverage type to a single exposure type. Sometimes, you need to link a coverage to several exposure types. For example, you want liability coverage on a claim to allow exposures for injured people and damaged property.

Prevent the Generator from adding CoverageSubtype typecodes

Before you begin

1. “Run the ClaimCenter Typelist Generator” on page 486 and generate the typelists
2. “Copy generated files to ClaimCenter” on page 489

About this task

The ClaimCenter Typelist Generator automatically creates typecodes in the `CoverageSubtype` typelist. There may be specific typecodes that you do not need in ClaimCenter. Follow these instructions to prevent the Generator from generating specific typecodes. This process requires that you first run the Generator and copy generated files to ClaimCenter.

Procedure

1. In Studio for ClaimCenter, open the `CoverageSubtype` typelist.
2. Click **Text** to view the XML version of the typelist.
3. Find the typecode on which you do not wish to generate coverage subtypes.
4. Surround the typecode with XML comments.
This sample XML comments out a typecode:

```
<!--  
<typecode ...>...</typecode>  
-->
```

Linking PolicyCenter coverages to the ClaimCenter general damage exposure type

To configure the ClaimCenter Typelist Generator to associate new coverages from PolicyCenter with the General Damage exposure type in ClaimCenter, run the generator with the `-Dmap_coverages=true` option.

When enabling this option, the typelist file `ExposureType.ttx` must be included in the input directory. The generator updates the `GeneralDamage` typecode by adding new coverage subtypes as categories.

If you configure the generator to link new coverages to `GeneralDamage`, you can demonstrate intake of First Notice of Loss (FNOL) without further configuration in Guidewire Studio for ClaimCenter. ClaimCenter displays a generic exposure page, which lets users open new claims against policies with the new coverages.

In a production environment however, Guidewire recommends that you map coverages to more specific exposure types in Guidewire Studio for ClaimCenter. Do not configure the generator to link new coverages to the General Damage exposure type if you plan to use more specific exposure types. Otherwise, you must find and delete links to the General Damage before you create new links to correct exposure types.

Preserving third-party claim system codes in generated typelists

PolicyCenter uses the source system category properties of codes in ClaimCenter typelists to determine the origin of the codes. A value of PC indicates ClaimCenter codes that originate in PolicyCenter. The ClaimCenter Typelist Generator adds, changes, or deletes only codes that originate in PolicyCenter.

Any value for source system category other than PC, including the absence of a source system category, indicates ClaimCenter codes that originate somewhere other than PolicyCenter. Those typecodes pass through the generator unchanged from input to output.

Merging PolicyCenter localization with ClaimCenter localization

If you configure your PolicyCenter and ClaimCenter instances with multiple locales, the ClaimCenter Typelist Generator helps you merge PolicyCenter localization with ClaimCenter localization.

Run the ClaimCenter Typelist Generator

To provide ClaimCenter with new product model information from PolicyCenter, you must generate typelist files that contain the typecodes for that information.

Before you begin

Before you run the ClaimCenter Typelist Generator, the following conditions must exist.

- You must know the location of the input and output directories.
- You must place the ClaimCenter typelist and typelist localization files in the input directory.

About this task

The generator will preserve the lines of business codes that you configured in Guidewire Studio for ClaimCenter or imported from other third-party policy administration systems.

Procedure

1. Copy the following files from `ClaimCenter/modules/configuration/config/extensions/typelist` to your input directory:
 - `CoverageSubtype.ttx`
 - `CoverageType.ttx`
 - `CovTermPattern.ttx`
 - `ExposureType.ttx`
 - `LOBCode.ttx`
 - `LossPartyType.ttx`
 - `PolicyType.ttx`
2. Copy files named `typelist.properties` and `typelist_<LanguageName>.properties` from the ClaimCenter locale directory to the input directory.

The `<LanguageName>` in the file name is the localization code. ClaimCenter localization files are located in the following directory.

```
ClaimCenter/modules/configuration/config/locale/
```

For example, if your instances have three locales, Canadian English, US English, and Canadian French, copy the following files to the input directory.

- `typelist_en_CA.properties`
- `typelist.properties`
- `typelist_fr_CA.properties`

3. At a command prompt, navigate to the `PolicyCenter` directory.

4. Run the following command.

```
gwb ccTypelistGen -Dinput_dir=input_dir -Doutput_dir=output_dir -Dmap_coverages={true|false}  
-Dcc_app_version={9|10}
```

The following table lists the command arguments. These arguments do not have default values. You must provide a value for every argument.

Argument	Value
-Dinput_dir	The directory that contains the input typelist files.
-Doutput_dir	The directory in which to place the output typelist files.
-Dmap_coverages	Whether to map new PolicyCenter coverages to the ClaimCenter generic General Damage exposure type. Specify true or false. This option is particularly helpful during the initial development phase of a configuration.
-Dcc_app_version	The ClaimCenter major version relevant to the input and output LOB typelists. Specify 9 or 10.

5. Check the output directory for generated typelist and localization files and the generation report `productModelGenReport.txt`.

In the output directory, the generator creates `typelist_LL_CC.properties` localization files for each locale.

For example, if your instances have three locales, Canadian English, US English, and Canadian French, the generator creates the following files.

- `OUTPUT_DIRECTORY/typelist_en_CA.properties`
- `OUTPUT_DIRECTORY/typelist.properties`
- `OUTPUT_DIRECTORY/typelist_fr_CA.properties`

6. Use the generation report to achieve the following goals.

- Determine success or failure of the generation command.
- If the command succeeded, identify new coverage types to map to exposure types in Guidewire Studio for ClaimCenter.

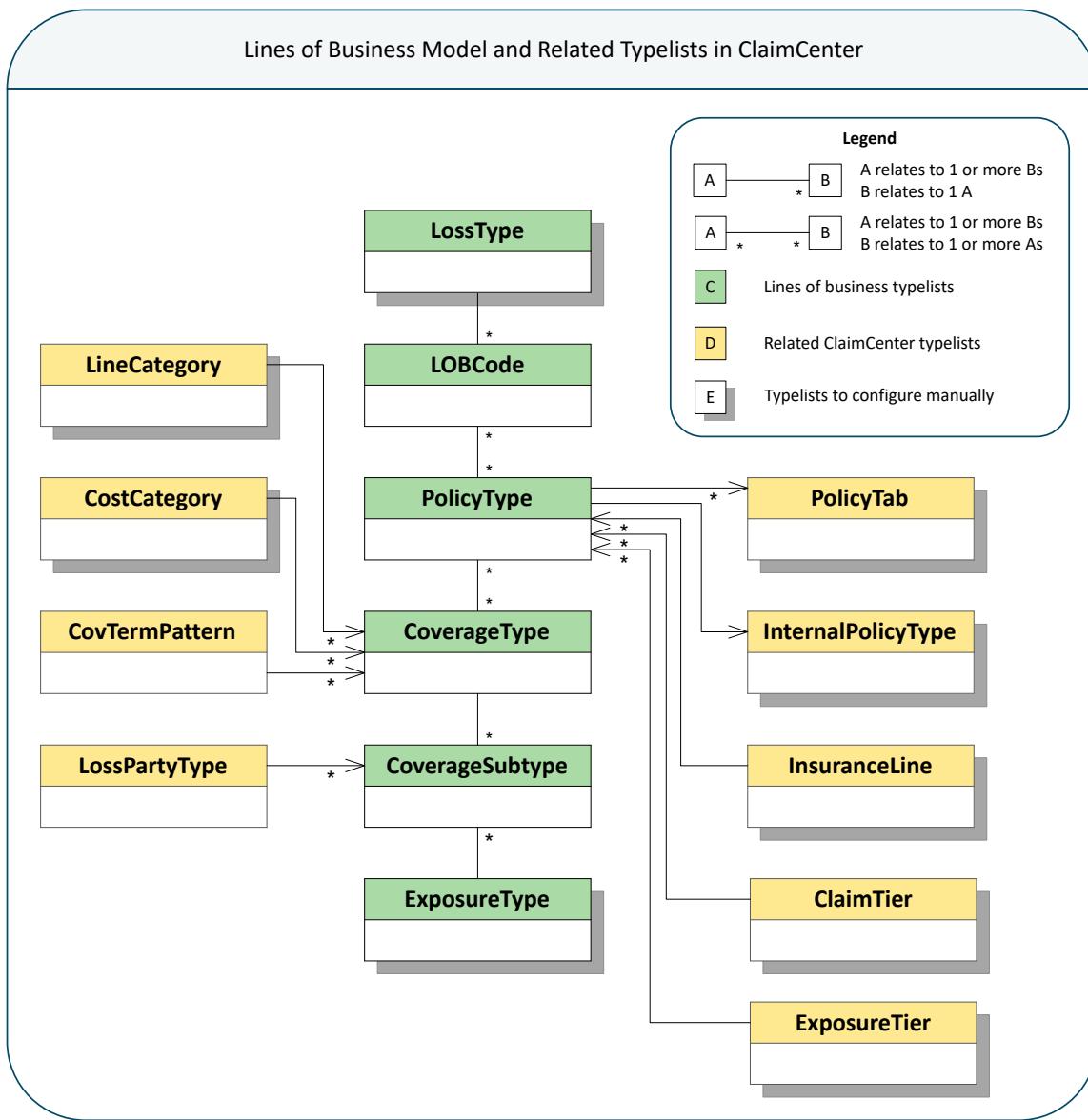
What to do next

To complete the integration of new lines of business into the ClaimCenter model, you perform the following tasks.

- “Copy generated files to ClaimCenter” on page 489
- “Link new lines of business types to loss types” on page 490
- “Link new coverage types to an exposure type” on page 490
- “Adding new lines of business codes to related ClaimCenter typelists” on page 492
- “Adding references to codes in Gosu classes and configuration files” on page 492

Using generated typelists in ClaimCenter

After you run the ClaimCenter Typelist Generator, you must perform additional steps to merge the changes from PolicyCenter into your ClaimCenter lines of business model. The following diagram highlights the typelists that you configure for this purpose in Guidewire Studio for ClaimCenter.



The following list shows suggested operations to perform.

1. Copy the generated typelist files and the generated typelist localization files to the ClaimCenter directories.
2. Use the generation report `productModelGenReport.txt` to determine the coverage types and LOB codes that the generator added.
3. If necessary, sort the XML elements and attributes in the input and output files to ensure that the files can be compared easily. Then, use a difference-detection tool on your input and output typelist files to determine any other typecodes that the generator changed, such as policy types.
4. In Guidewire Studio for ClaimCenter, link generated lines of business typecodes to the top of the lines of business model. For new LOBCode typecodes, add appropriate `LossType` typecodes as parents.
5. In Guidewire Studio for ClaimCenter, link generated lines of business typecodes to the bottom of the lines of business model. For new `CoverageSubtype` typecodes, add appropriate `ExposureType` typecodes as children.
6. In Guidewire Studio for ClaimCenter, link generated lines of business codes to related ClaimCenter typelists.
7. If you add or remove `CoverageSubtype` typecodes, run the generator again with your latest ClaimCenter typelist files. The generator adjusts `LossPartyType` for you.

8. Add references to new codes in ClaimCenter Gosu classes and other configuration files that relate to coverages types and policy types.
9. Search ClaimCenter for references to lines of business codes that the generator removed, such as references in rules. Fix obsolete references by deleting them or changing them to use other codes.

Copy generated files to ClaimCenter

You place the files that contain new product model information from PolicyCenter into the correct locations for ClaimCenter to access the information.

About this task

After you run the ClaimCenter Typelist Generator, you copy the generated files to ClaimCenter. You do not perform the first step if you configured the generator to use the ClaimCenter directory for lines of business typelist files as its input and output directory.

Procedure

1. If necessary, copy generated typelist files (.ttx) to:

```
ClaimCenter/modules/configuration/config/extensions/typelist
```

2. Copy the typelist localization (.properties) files from the output directory to the ClaimCenter locale directory.

```
ClaimCenter/modules/configuration/config/locale
```

Using the generation report to identify added coverages and LOB codes

The ClaimCenter Typelist Generator writes a generation report, `productModelGenReport.txt`, in the output directory. The report includes lines for the following items.

- Coverage subtypes that the generator added to `CoverageType` but did not link to exposure types in `ExposureType`.
- LOB codes that the generator added to `LOBType` but did not link to loss types in `LossType`.

The following example shows lines from the generation report.

```
...
Warning: LOB Code [BOPLine] is not mapped to any loss types.
Warning: LOB Code [GLLine] is not mapped to any loss types.
Warning: LOB Code [BusinessAutoline] is not mapped to any loss types.
Warning: LOB Code [GolfCartLine] is not mapped to any loss types.
...
Warning: Coverage subtype [BOPBuildingCov] is not mapped to any exposure types.
Warning: Coverage subtype [BOPOrdinanceCov] is not mapped to any exposure types.
Warning: Coverage subtype [BOPPersonalPropCov] is not mapped to any exposure types.
Warning: A new default coverage subtype was created for [zfhg4jesa2eia1ht9ie8ggceba8].
    It is not mapped to any exposure types.
Warning: A new default coverage subtype was created for [z98j04d2c97fv95ebf7qh3qc4rb].
    It is not mapped to any exposure types.
...
```

Values in square brackets are typecodes that were not linked. In Guidewire Studio for ClaimCenter, you must link reported LOB codes to loss types, and you must link reported coverage subtypes to exposure types.

If you chose to link new coverages to the General Damage exposure type when you ran the generator, the generation report does not include lines for new coverage types. The generator linked all new coverage types to the typecode `GeneralDamage` in `ExposureType`. In this case, use a difference tool to compare the input and output versions of `CoverageSubtype.ttx` to identify new coverage types that you must link to exposure types.

Link new lines of business types to loss types

In ClaimCenter, you must link the new product model information from PolicyCenter to the appropriate loss types.

Before you begin

Before you can link new business types to loss types, you must perform the following tasks.

- “Run the ClaimCenter Typelist Generator” on page 486
- “Copy generated files to ClaimCenter” on page 489

About this task

Studio displays errors for new lines of business that you create by using the ClaimCenter typelist generator because the generator cannot add loss types to the line of business. Each line of business must have a parent loss type. To clear the error conditions, you add each new line of business type to a loss type.

Procedure

1. In Guidewire Studio for ClaimCenter, navigate to **configuration > config > Extensions > Typelist**, and open the **LossType.ttx** file.
2. On the **Line of Business** tab, in the list in the **LOB** column, select the loss type that you want to assign to the new coverage type.
3. Right-click the loss type, and then click **Add new > LOBCode**.
4. To add the new line of business type to this loss type, click the **Select typecode** tab.
 - a) Select the **Typecode** from the list of unassigned typecodes.
 - b) Click **OK**.

Results

The new line of business is linked to the loss type.

What to do next

To complete the integration of new lines of business into the ClaimCenter model, you perform the following tasks.

- “Link new coverage types to an exposure type” on page 490
- “Adding new lines of business codes to related ClaimCenter typelists” on page 492
- “Adding references to codes in Gosu classes and configuration files” on page 492

Link new coverage types to an exposure type

In ClaimCenter, you must link new product model information from PolicyCenter to an exposure type.

Before you begin

Before you can link new coverage types to exposure types, you must perform the following tasks.

- “Run the ClaimCenter Typelist Generator” on page 486
- “Copy generated files to ClaimCenter” on page 489
- “Link new lines of business types to loss types” on page 490

About this task

If the generator linked new coverages to the General Damage exposure type, you can change this exposure type. If the generator did not link new coverages to the General Damage exposure type, Studio displays errors because each coverage must have an exposure type. To clear the error conditions, you add an exposure type to each new coverage subtype.

Procedure

1. In Guidewire Studio for ClaimCenter, navigate to **configuration > config > Extensions > Typelist**, and open the **CoverageSubtype.ttx** file.
2. On the **Line of Business** tab, in the list in the **LOB** column, select the new coverage subtype.
3. If the generator linked new coverages to the General Damage exposure type, expand the coverage subtype and the **Children** folder, right-click the row for **GeneralDamage**, and then click **Remove**.
4. Right-click the coverage subtype, and then click **Add**.

To create a new exposure type for this subtype, use the **New typecode** tab.

- a) Enter a value for **Code** and **Name**.
- b) Optionally, enter a value for **Description** and **Priority**.
- c) Select an **Incident**.
- d) Click **OK**.

To use an existing exposure type for this subtype, use the **Select typecode** tab.

- a) Select a **Typecode** from the list of existing typecodes.
- b) Click **OK**.

Results

The new coverage is linked to the exposure type through its generic coverage subtype. In the following screenshot from Guidewire Studio for ClaimCenter, **New Coverage Subtype** links to the ClaimCenter Property exposure type.

The screenshot shows a software interface for managing typelists. The window title is "CoverageSubtype.ttx". The main area is a table with three columns: LOB, Typecode, and Typelist. The rows show the structure of the typelist:

LOB	Typecode	Typelist
Coverage for Injury to Leased Workers	GLAddInjuryLeasedWorkers	CoverageSubtype
Death & Disability Benefit	z98j04d2c97fv95ebf7qh3qc4rb	CoverageSubtype
Children		ExposureType
Bodily Injury	BodilyInjuryDamage	ExposureType
Parents		CoverageSubtype
Other Categories		
InjuryIncident	InjuryIncident	Incident
Parents		CoverageType
Other Categories		
PC	PC	SourceSystem

What to do next

To complete the integration of new lines of business into the ClaimCenter model, you perform the following tasks.

- “Adding new lines of business codes to related ClaimCenter typelists” on page 492
- “Adding references to codes in Gosu classes and configuration files” on page 492

Adding new lines of business codes to related ClaimCenter typelists

To complete the integration of new lines of business into the ClaimCenter model, you perform the following general tasks:

1. For new PolicyType typecodes, add PolicyTab typecodes as categories.
2. For new PolicyType typecodes, add one InternalPolicyType typecode, commercial or personal, as a category.
3. For new PolicyType typecodes, add them as categories to appropriate InsuranceLine, ClaimTier, and ExposureTier typecodes.
4. For new CoverageType typecodes, add them as categories to appropriate LineCategory and CostCategory typecodes.

Note that the generator writes an updated CovTermPattern.ttx and LossPartyType.ttx files, so you do not need to change CovTermPattern or LossPartyType in Guidewire Studio for ClaimCenter.

Adding references to codes in Gosu classes and configuration files

For ClaimCenter to take full advantage of new codes from PolicyCenter, you must add references to new codes in page configurations (.pcf), Gosu classes (.gs), and other configuration files.

The following notes apply to changing new coverages.

- ClaimCenter accepts data in ACORD format and maps the data to new claims. Review the following files for two places to add ACORD mappings for new coverage types.
 - In **configuration > gsrc, gw.fnolmapper.acord.impl.AcordExposureMapper.gs**
 - In **configuration > config, fnolmapper.acord.typecodemapping.xml**
- To support ISO Claim Search for new coverage types, add entries to the following CSV text file.
 - In **configuration > config, iso.ISOCoverageCodeMap.csv**
- ClaimCenter has a number of methods that categorize PIP coverages. These methods govern the display of benefits tabs in ClaimCenter. For new or removed PIP coverage types, review the following Gosu class.
 - In **configuration > gsrc, libraries.PolicyUI.gsx**

The following notes apply to changing new policy types.

- Review the programming logic for exposure tier and claim tier mapping. You may need to change the logic in the following Gosu classes for new policy types.
 - In **configuration > gsrc, gw.entity.GWClaimTierEnhancement.gsx**
 - In **configuration > gsrc, gw.entity.GWExposureTierEnhancement.gsx**
- Review the programming logic for setting initial value on new claims. ClaimCenter generally sets the LOB code on new claims based on the loss type for the claim. You may need to change the logic in the following Gosu function.
 - In **configuration > gsrc, libraries.ClaimUI.setInitialValues**
- To enforce aggregate limits and policy periods for new policy types, review the settings in the following files.
 - In **configuration > config, aggregatelimit.aggregateLimitUsed-config.xml**
 - In **configuration > config, aggregatelimit.policyPeriod-config.xml**

Typelist localization

If you configure your PolicyCenter and ClaimCenter instances with multiple locales, the ClaimCenter Typelist Generator helps you with typelist localization. The generator produces an updated typelist localization file for each

locale. A typelist localization file contains translated typecode names and descriptions for a specific locale for all typelists in a Guidewire instance. ClaimCenter locale files are located in the following directory.

```
ClaimCenter/modules/configuration/config/locale/
```

The generator produces updated ClaimCenter localization files with names and descriptions for new typecodes that come from PolicyCenter. The ClaimCenter Typelist Generator always produces a localization file for the default locale in your PolicyCenter instance, regardless of whether you configure the instance for multiple locales.

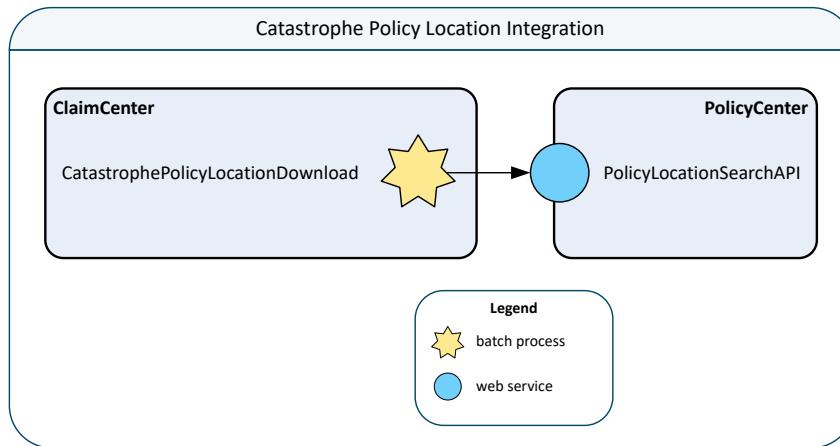
The generator preserves names and descriptions for typecodes that do not originate in PolicyCenter. The generator does not remove names and descriptions for typecodes that originated in PolicyCenter and now are deleted. The generator cannot distinguish localization properties that refer to deleted PolicyCenter typecode and properties for typecodes that originate from elsewhere. Localized strings for typecodes that are removed from PolicyCenter remain in the ClaimCenter typelist localization files.

See also

- “Run the ClaimCenter Typelist Generator” on page 486
- “Copy generated files to ClaimCenter” on page 489

Catastrophe policy location download

ClaimCenter provides the Catastrophe Policy Location Download batch process to retrieve policy locations from PolicyCenter or other, third-party policy administration systems. The batch process requests policy locations that lie within a geographic area of interest that ClaimCenter administrators set for a designated catastrophe.



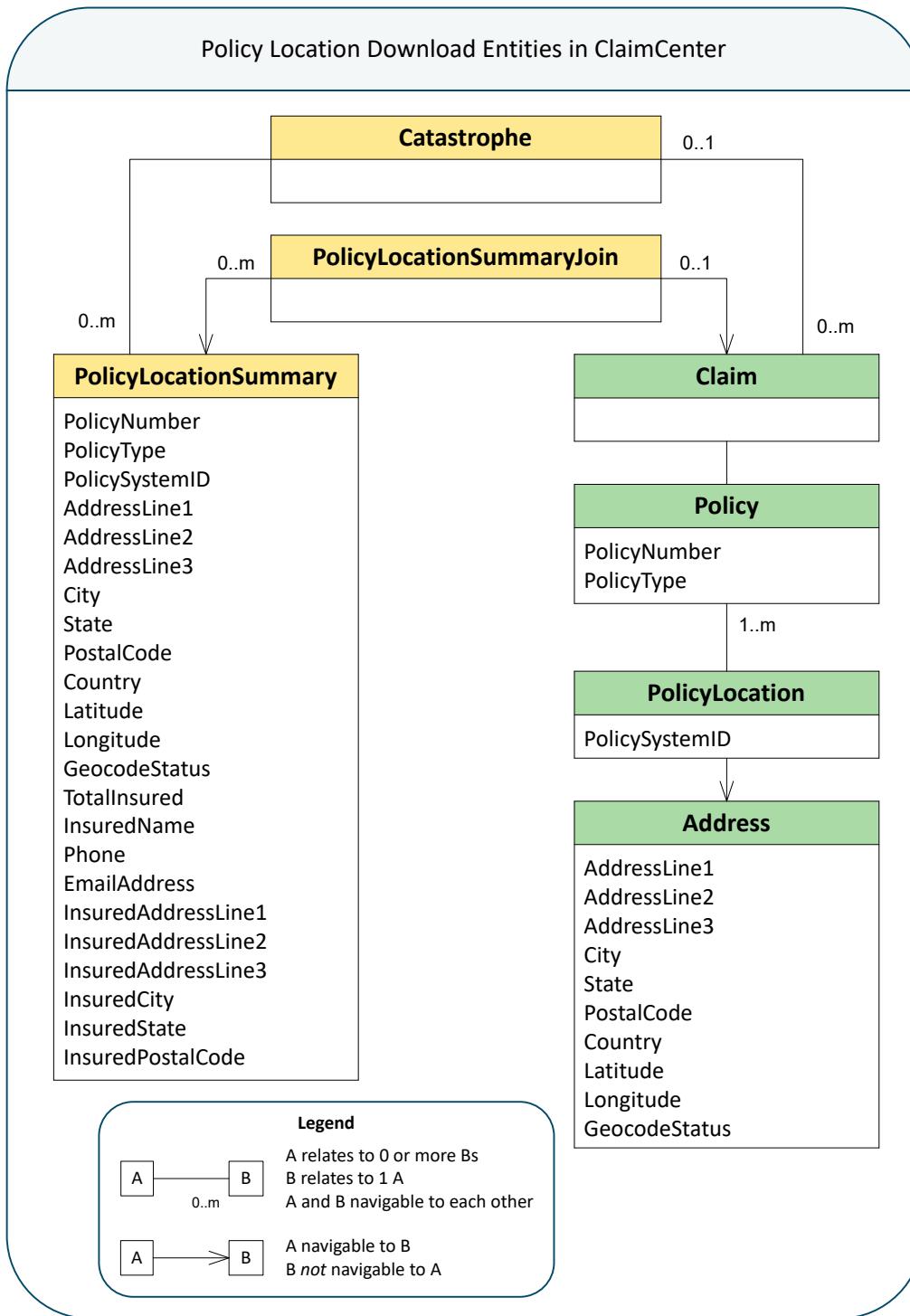
If you integrate PolicyCenter with ClaimCenter, the default implementation of the `CatastrophePolicyLocationDownload` batch process calls the `PolicyLocationSearchAPI` web service that PolicyCenter publishes. The batch process is written entirely in Gosu, so you can modify the default implementation to download policy locations from third-party policy administration systems.

Catastrophe policy location overview

In ClaimCenter, administrators define catastrophes. To specify the area of interest for a catastrophe, an administrator clicks **Edit**, drags a bounding box on the catastrophe map, and then clicks **Set Catastrophe Area of Interest**. The next time that the batch process runs, ClaimCenter downloads policy locations from a policy administration system based on the area of interest and the effective date of the catastrophe. After the batch process downloads a fresh set of policy locations for the catastrophe, users who view the catastrophe map can see them plotted on the catastrophe map.

Catastrophe areas of interest data model

The `CatastrophePolicyLocationDownload` batch process uses several entities in the ClaimCenter data model.



The batch process creates a `PolicyLocationSummaryJoin` to associate a `PolicyLocationSummary` and a `Claim` if all the following conditions are true.

- The `Claim.LossLocation.Address` exactly matches the address fields on a `PolicyLocation.Address` in the `Claim.Policy.PolicyLocations` array. The geocode fields `Latitude` and `Longitude` are not used for matching.
- The `PolicyLocationSummary.PolicyNumber` equals the `Claim.Policy.PolicyNumber`.
- The `PolicyLocationSummary.PolicySystemID` equals the `PolicyLocation.PolicySystemID` of the matching `PolicyLocation` in the `Claim.Policy.PolicyLocations` array.

The default Gosu implementation of the batch process does not verify that the address fields on the `PolicyLocationSummary` exactly match the address fields on the matching `PolicyLocation.Address` in the `Claim.Policy.PolicyLocations` array. You can modify the Gosu implementation to add this additional consistency check.

The policy location summary entity

The catastrophe heat map in ClaimCenter plots policy locations based on `PolicyLocationSummary` instances. A `PolicyLocationSummary` instance contains information about a policy location, regardless of any claim filed against it in ClaimCenter. The catastrophe heat map does not use the `Claim.Policy.PolicyLocations` array to plot policy locations.

The batch process `CatastrophePolicyLocationDownload` calls the `PolicyLocationSearchAPI` web service to download policy location information from PolicyCenter. The batch process uses the downloaded information to create `PolicyLocationSummary` instances, as well as to create `PolicyLocationSummaryJoin` instances when it finds matching `Claim` instances. Besides the batch process, ClaimCenter creates and maintains `PolicyLocationSummaryJoin` instances in response to updated `Claim` instances that match a `PolicyLocationSummary` instance.

The important properties of the `PolicyLocationSummary` are described in the following list.

- `PolicyNumber` – Policy number from the external policy administration system, such as PolicyCenter
- `PolicySystemID` – Identifier of the external policy administration system, such as PolicyCenter
- Policy location address – Names of location address properties are the same as on the `Address` entity
- `Latitude`, `Longitude`, and `GeocodeStatus` – Geocode attributes for the policy location address
- Insured's contact details, including the following information.
 - `InsuredName`
 - `Phone`
 - `EmailAddress`
 - `Insured*` – Names of the insured's address fields begin with `Insured`.
- `TotalInsured` – Total insured value for the policy location, as a currency amount.

The policy location summary join entity

A `PolicyLocationSummaryJoin` instance associates a `PolicyLocationSummary` with a matching catastrophe `Claim`. Usually, significantly more policy location summaries exist than matching catastrophe claims.

The `PolicyLocationSummaryJoin` entity associates `Claim` and a `PolicyLocationsSummary`. The entity has two properties, both required.

- `PolicyLocationSummary` – a foreign key to a `PolicyLocationSummary`.
- `Claim` – a foreign key to a matching catastrophe `Claim`.

These two foreign keys serve the only purpose of a `PolicyLocationSummaryJoin` instance, associating a `PolicyLocationSummary` with a matching catastrophe `Claim`.

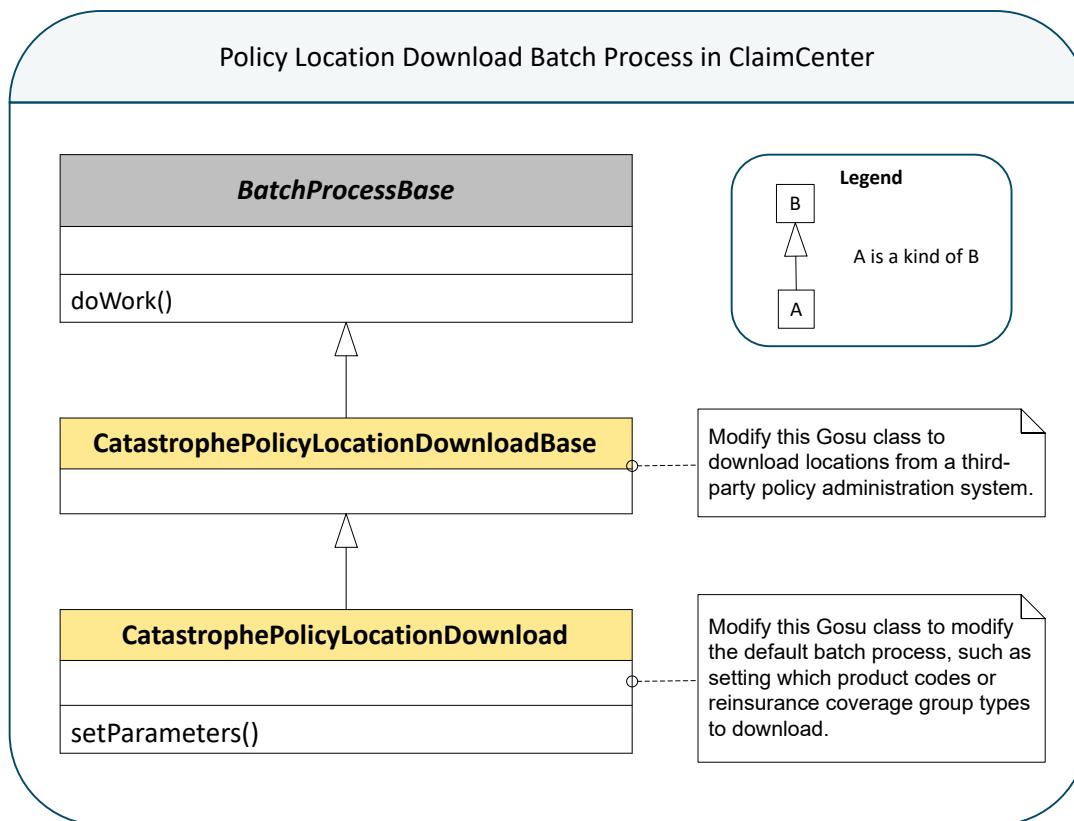
Catastrophe policy location download processing

When it runs, the Catastrophe Policy Location Download batch process looks for catastrophes where the last modified date is later than the date of the last download of policy locations. For each catastrophe that qualifies, the batch process updates the policy locations in the modified area in two phases.

1. Location Query Phase – Queries PolicyCenter or a third-party policy administration system for locations within the area of interest through the `PolicyLocationsSearchAPI` web service that PolicyCenter publishes.
2. Claim Matching Phase – Matches policy locations downloaded from the policy administration system with policies on claims that already are in ClaimCenter.

Catastrophe policy download batch process implementation classes

Like all batch processes, `CatastrophePolicyLocationDownload` extends the `BatchProcessBase` abstract class. It does so indirectly by extending `CatastrophePolicyLocationDownloadBase`, which in turn extends `BatchProcessBase` directly.



Generally, you modify the `CatastrophePolicyLocationDownload` class to modify the default batch process. For example, modify the `setParameters` method to set the arrays or product codes or reinsurance coverage group types to download. The default product codes are `Homeowners` and `CommercialProperty`. The default reinsurance coverage group type is `Property`. `PolicyCenter` defines its full set of reinsurance coverage group types with the `RICoverageGroupType` typelist.

To download locations from a third-party policy administration system, modify the `CatastrophePolicyLocationDownloadBase` class.

Catastrophe policy download batch process error handling

The default implementation of the batch process attempts to connect with `PolicyCenter` through its `PolicyLocationSearchAPI` web service. If the web service is unavailable when the batch process runs, it fails. However, the batch process retries to connect to the web service the next time the batch process runs.

Policy location search plugin

To perform a policy location search, `ClaimCenter` calls methods of the `PolicyLocationSearchPlugin` plugin interface. Each return result has properties to represent the policy number, the address, the latitude, the longitude, product code, and insured value.

`ClaimCenter` include a base implementation of the plugin interface that makes a web service call to `PolicyCenter` to search policy locations.

```
gw.plugin.pcintegration.pcVERSION.location.PolicyLocationSearchPluginImpl
```

The base implementation uses the following related class for encapsulating the return results.

```
gw.plugin.pcintegration.pcVERSION.location.CCPolicyLocationInfoImpl
```

If you use PolicyCenter, you may want to edit those files to make your own modifications, or base a new version of the plugin on this version. If you use a different policy system, write your own implementation of this plugin.

Next, update the plugin registry in Studio to use your implementation instead of the default plugin implementation.

There is only one method in this interface:

```
findPolicyLocationsWithinBoundingBox(criteria : PolicyLocationSearchCriteria) : CCPolicyLocationInfo[]
```

The method finds policy locations within a bounding box, as well as other search criteria defined by `PolicyLocationSearchCriteria`. The following properties are included.

- `EffDate` – Effective date, as `Date`
- `ProductCodes` – Product codes, as an array of `String` objects
- `TopLeftLat`, `TopLeftLong`, `BottomRightLat`, `BottomRightLong` – Bounding box in latitude and longitude
- `Handle` – The `String` handle, which is used for returning multiple requests of results for the same query
- `StartingOffset` – The `int` starting offset in the results for large result sets
- `Count` – The number of items to return in this search

In your version of the `findPolicyLocationsWithinBoundingBox` method, use the properties in the `PolicyLocationSearchCriteria` to initiate a query in the external policy system.

To support large result sets and paged results, the plugin supports multiple queries to the same result set using a unique value called the handle. The handle is in the search criteria object as the `Handle` property. The handle identifies a single persistent query. The `StartingOffset` number in the search criteria is the index within the result set to start returning more results. ClaimCenter increases the value of `StartingOffset` on each request to get the next batch of results.

In the default implementation, PolicyCenter retrieves all results in memory on the first request with that handle value. PolicyCenter caches the results and reuses them for additional requests with the same handle value.

If you write your own implementation of this plugin, follow a similar pattern. Cache search results and store the handle value on external system so subsequent search queries with the same handle return results quickly.

Return the results in an array of `gw.plugin.policy.location.CCPolicyLocationInfo` objects. The `CCTPolicyLocationInfo` interface defines the set of properties on the object. In Studio, you can view the interface directly. Alternatively, in Studio, look at the `CCTPolicyLocationInfoImpl` implementation class mentioned earlier for how to implement that class in Gosu and to review the list of properties on the results.

Policy refresh overview

Each claim includes a copy of the associated policy that was in force on the date of loss. This policy was retrieved from the external policy administration system. At a later time, you can update the policy from the external policy administration system. For example, if you change the date of loss, there may be different policy coverages in force on that updated date. At the end of policy refresh, the older copy of the policy is deleted entirely. This entire process is called policy refresh. The copy of the policy in ClaimCenter is called a policy snapshot and often is a subset of the policy information in the policy administration system.

Preserving relationships among claims and policies with policy refresh

The technical challenge of policy refresh logic is to preserve the relationship among the claim and the policy. Think of the policy as a subgraph of the claim graph. Various objects within the claim graph might have links directly to objects within the policy graph. For example, an exposure object in the claim has a foreign key to a coverage in the policy. When you refresh a policy, ClaimCenter replaces policy-specific entities with new ones retrieved from the policy administration system. However, there may be important differences between the old copy of the policy and the newer copy of the policy.

- There could be new objects, such as a new claim contact (`ClaimContact`).
- There could be changed objects, such as changed coverage details.
- There could be objects with no equivalent in the new policy, such as a coverage that no longer exists. If this happens, ClaimCenter deletes the object by the end of policy refresh, assuming there are no errors that would prevent it.

At a technical level, the refresh process includes the following high-level steps.

1. ClaimCenter determines the set of entities in the policy graph.
2. ClaimCenter compares the original copy of the current policy with the new policy and determines which current policy objects correspond to which objects in the new policy. The comparison happens using a set of classes called entity matchers. An entity matcher knows how to compare two policy objects of the same entity type to determine whether they logically represent the same thing. For example, if there is only one contact in both the current policy and the new policy, are they the same person or an entirely different person?
Be sure you understand matching before modifying or writing any matching code. If you do not match and relink objects successfully, you could corrupt the claim/policy graph.
3. ClaimCenter displays important differences between the current policy and the new policy. If you confirm the policy refresh, you know which policy objects ClaimCenter will add, update, or delete. You can also view any warnings or errors for policy refresh. An example of a warning is that the effective dates on a coverage changed. An example of an error is that there exists a payment on a coverage that no longer exists. In contrast to warnings, any error blocks policy refresh from completing. If there are no errors, you can confirm the policy refresh. You can configure warnings and errors in Gosu as part of the difference display configuration.
4. ClaimCenter proceeds to replace the policy. A critical part of this is relinking objects between the claim and the policy. Relinking requires fixing foreign keys from a claim object to a policy object, or from a policy object to a claim object. It is critical that ClaimCenter fixes any links that are potentially broken when replacing the current policy graph with a new policy graph. For example, an exposure with a foreign key to a policy coverage must link to the newer copy of the policy coverage. ClaimCenter encapsulates critical parts of relinking logic in objects called relink handlers. A relink handler defines the behavior when a foreign key between a policy object and a claim object might break during policy refresh. The default behavior depends on direction of the broken link, and whether the object exists on the new policy.
Be sure you understand relinking before modifying or writing any relinking code. If you do not match and relink objects successfully, you could corrupt the claim/policy graph.
5. ClaimCenter purges the old policy objects from the database. ClaimCenter deletes the database rows, rather than retiring them. Even if the object did not change between the old and new policy, the old object is deleted because the new object completely replaces the old object. Because relinking already finished, any foreign keys to the old object already link to the new copy of the object.
6. Finally, all cumulative changes during policy refresh commit to the database in one database transaction.

See also

- “Determining the extent of the policy graph” on page 504

Matching overview

For some objects matching is easy if the object has another property that is both uniquely identifying and immutable (unchangeable). For example, a contact has a Address Book Unique ID (ABUID) property. If the ABUID values match, it is a definitive match.

For other entities, this is more complex and requires testing multiple properties. For example, a vehicle’s matcher can match the vehicle identification number (VIN) if it is present. Otherwise, the matcher compares other properties such as the license plate number, the state, or the vehicle serial number.

A small number of entity types match based on matching related objects, such as parent objects. For example, the default behavior is to match two `CovTerm` objects based on having matching parent `Coverage` objects.

For some but not all policy objects, there is a field containing a policy object `Policy System ID` (PSID) in the `PolicySystemId` property. The PSID is a unique ID that your external policy administration system specifies as

uniquely representing the object in the policy administration system. If two objects have the same entity type and a matching PSID, it is a definitive match. The policy administration system must ensure that the PSID is both unique and immutable.

Some entities do not have the `PolicySystemId` property. Usually those entities already have another uniquely identifying immutable property, for example contacts have the `ABUID` in its `AddressBookUID` property. However, some entities might have multiple types of unique immutable IDs.

If you add new entity types that add identifying fields to the subtype, write a new matching class for each new entity type. You must do this even if you use a PSID to match that entity type. However, if you add new entity types that do not add identifying fields to the subtype, you do not need to write a new matching class for the subtype.

Differences between policy refresh and policy select

If you select an entirely different policy from the user interface (the process called policy select), a similar policy refresh happens.

There are differences between a policy refresh and a policy select.

- With policy refresh, the policy represents the same policy though it may have changed since last downloaded.
- With policy select, the entire policy including the policy number and the entire graph might be completely different. Although both policy refresh and policy both set a flag that indicates the policy is verified, policy select is the recommended approach for verifying a policy. For example, the old policy may have been entered manually with incomplete or incorrect information, perhaps because the external policy administration system was offline when originally entered.

However, from a technical standpoint for implementing policy refresh integration, the implementation is the same. In both cases, ClaimCenter calls the policy refresh plugin and the behavior of the plugin typically would be basically the same.

How policy refresh handles special situations

The following list describes the policy refresh behavior in the ClaimCenter base configuration.

Retrieved policy	During policy refresh...
Contact no longer present on the policy as insured, and is added as an excluded party	The insured becomes the former insured. ClaimCenter adds the contact as an excluded party
Coverage added	ClaimCenter shows the refreshed policy with the coverage.
Coverage incident or exposure limits changed (or the limit currency changed)	ClaimCenter updates the coverage limits on the policy. ClaimCenter provides a warning if the limit decreases.
Effective date or the expiration date changed on the coverage	ClaimCenter provides a warning.
PIP aggregate limits lowered	ClaimCenter provides a warning.
Policy currency changed	<ul style="list-style-type: none">If the currency is editable then ClaimCenter provides a warning.If the currency is read-only then ClaimCenter blocks the refresh with an error.
Policy period changed	ClaimCenter provides a warning.
Risk unit added to the policy (a vehicle, for example)	ClaimCenter adds the risk unit to the claim. It is possible to modify an incident to use the new risk unit (the vehicle).
Risk unit coverage removed that is used by an exposure	<ul style="list-style-type: none">If the exposure is still open, ClaimCenter blocks the refresh. The user must close the exposure first to continue.

Retrieved policy	During policy refresh...
	<ul style="list-style-type: none"> • If there are non-reserving transactions on the exposure, or the net incurred is greater than zero, then ClaimCenter blocks the refresh. • If the exposure is closed, meaning that there are no transactions except reserves, and the net incurred is zero, ClaimCenter allows the refresh.
Spelling of an insured name changed	<p>The exact action is dependent on the matching logic.</p> <ul style="list-style-type: none"> • If the contact is uniquely identified, ClaimCenter treats the change as a name change. • If no unique identification is present, the default behavior in the base configuration is to use the name as an identifier, by using fallback matching criteria. ClaimCenter considers this change to be the addition of a new contact.
Workers' comp class code removed	ClaimCenter blocks the refresh with an error.

Policy refresh plugins and configuration classes

ClaimCenter defines a policy refresh plugin interface called `IPolicyRefreshPlugin`. This plugin defines the interactions between ClaimCenter and the policy refresh logic.

If you use PolicyCenter, use the built-in plugin implementation called `PCPolicyRefreshPlugin`. This class implements default behavior for the PolicyCenter data model.

If you use a policy administration system other than PolicyCenter, write a new plugin implementation. Base your plugin implementation on the included plugin implementation class `BasicPolicyRefreshPlugin`, which is implemented in Gosu. This class extends the internal base class `PolicyRefreshPluginBase`, which implements core behaviors of policy refresh. This internal base class is implemented in Java and is not visible in Studio.

In the default configuration, each built-in plugin implementation class relies on a policy refresh configuration class to handle nearly all of the work. This approach encapsulates all code related to policy refresh in one place rather than implementing it directly in a plugin implementation class. All the built-in configuration are written in Gosu and are visible in Studio.

The default configuration defines a base interface called `PolicyRefreshConfiguration`. The `PolicyRefreshConfiguration` interface defines the basic contract between a policy refresh plugin and its policy refresh configuration object. There are several included implementations of this interface, as well as a subinterface, that correspond to the different plugin implementations. The following table summarizes the plugin implementation classes and their configuration classes.

Class	Purpose	Configuration implementation class that it uses
<code>BasicPolicyRefreshPlugin</code>	<p>Integrate policy refresh with a policy administration system other than PolicyCenter. This is also called the basic example plugin.</p> <p>If you extend the PolicyCenter data model, you must modify the policy refresh plugin to identify any new entities.</p>	<p>Uses the configuration class <code>BasicPolicyRefreshConfiguration</code> in the same package. It extracts the set of policy-only entities from an existing claim-and-policy graph. The code walks foreign key/array references and determines which entities that are created in ClaimCenter through the <code>IPolicySearchAdapter</code> plugin. This defines the set of policy-specific entity instances.</p> <p>If you want to write your own configuration class to work with this plugin, base your class on a new implementation of <code>ExtendablePolicyRefreshConfiguration</code>, which extends its superinterface <code>PolicyRefreshConfiguration</code>. Override the <code>getPolicyOnly</code> method to specify the entities that are part of the policy graph.</p>
<code>PCPolicyRefreshPlugin</code>	Integrate policy refresh with Guidewire PolicyCenter.	Uses the configuration class <code>PCPolicyRefreshConfiguration</code> in the same package. This configuration object finds entities to replace during policy refresh by examining XSD types in the web service

Class	Purpose	Configuration implementation class that it uses
	If you extend the PolicyCenter data model, you do not need to modify the policy refresh plugin for PolicySearchPCPlugin. any new entities. Extension entities are automatically detected when ClaimCenter imports the web service that PolicyCenter publishes.	definition to connect to PolicyCenter. This plugin implementation relies on the built-in policy search plugin implementation for PolicyCenter: <code>gw.plugin.pcintegration.pcVERSION_NUMBER</code> .
<code>PolicyRefreshPluginBase</code>	The internal base class for the other two policy refresh <code>PolicyRefreshConfigurationBase</code> . This base class is plugin implementations not directly extend this class.	The plugin base class uses the configuration base class important because it implements the core functionality. Refer to it in that this table mentions. Do Studio.

You could use one of several different strategies to write a custom policy refresh plugin.

- For a policy administration system other than PolicyCenter, base your implementation on the built-in implementation `BasicPolicyRefreshPlugin`. Write an entirely new implementation of the `ExtendablePolicyRefreshConfiguration` interface, which is defined in Gosu and visible in Studio. Override the `getPolicyOnly` method to specify the entities that are part of the policy graph. In the plugin, override the constructor to return your configuration object implementation.
- For expert integration programmers only, you could choose to re-implement the interface entirely with your a completely new design. This approach is not recommended. If you completely reworked the plugin implementation, you could consider not using any of the built-in configuration object interfaces or classes.

You might not even need to change the plugin and configuration objects in a significant way. Instead you might need only to modify the more easily-customized parts of the policy refresh system.

- Entity matchers
- Difference display tree user interface

See also

- “Policy refresh entity matcher details” on page 505
- “Policy refresh policy comparison display” on page 513

Policy refresh configuration base class

The `PolicyRefreshConfigurationBase` abstract class controls almost all ClaimCenter interface configuration for policy refresh, either directly or indirectly. This class consists entirely of property getters, most of which define mappings between various entities and the classes used to manage those entities. To override these values, override property getters in your own subclass of the `PolicyRefreshConfigurationBase` class.

The following list describes the most important property getters in the `PolicyRefreshConfigurationBase` class.

Property getter	Used to...
<code>MatcherTypes</code>	Determine which entities in the existing and new policies are logically equivalent.
<code>DisplayTree</code>	Construct the policy comparison tree within ClaimCenter.
<code>DiffDisplayTypes</code>	Configure how ClaimCenter displays a specific entity type.
<code>CustomRelinkerTypes</code>	Determine how ClaimCenter relinks or fixes broken foreign key links that became broken as you refreshed a policy.

Property getter	Used to...
RelinkFilters	Construct special logic that is outside the standard Policy Refresh behavior.

Policy refresh steps and associated implementation

The following table describes the steps in a standard policy refresh. The rightmost column describes the location of the implementation for this step as well notes about the behavior.

#	Step Summary	Description	Implementation
1	In the ClaimCenter policy summary screen, click Policy Refresh or Policy Select .	To refresh the policy in ClaimCenter, click Policy Refresh . To verify the policy, click Policy Select .	See the ClaimCenter policy summary screen for PCF details.
2	Get new policy from external system	After the user selects to refresh the policy or select a new policy, ClaimCenter retrieves the new policy from the policy administration system.	ClaimCenter calls the policy search plugin (IPolicySearchAdapter) to find and retrieve the latest version policy using its retrievePolicyFromPolicy method.
3	Get policy-specific entities	ClaimCenter determines which entity instances references through the policy are policy-specific.	This step happens in the policy refresh plugin method called compare . That method returns results encapsulated as a PolicyComparison object (that is an interface name). In the base configuration, the policy refresh plugin instantiates the built-in class PolicyComparer . The PolicyComparer class in its extractPolicyGraph method returns a set of policy objects. In the default implementation, this object calls the configuration object's getPolicyOnly method. The configuration object getPolicyOnly method is the method you would typically override to modify the built-in behavior..
4	Find which objects to remove, add, or update	ClaimCenter compares the local existing policy and the new policy from the policy search looking for entities that were removed, added, or updated.	Sorts the entity instances from both policies into three sets: <ul style="list-style-type: none">• Objects removed from the existing policy• Objects added on the new policy• Objects that are logically the same on the two policies. In ClaimCenter terminology, these objects matched. This step happens in the policy refresh plugin method called compare . In the base configuration, the matching process is coordinated by an implementation of the PolicyRefreshConfiguration interface. Such a class extends the PolicyRefreshConfigurationBase Gosu class. PolicyRefreshConfigurationBase contains a mapping in its MatcherTypes property. That property specifies the mapping between: <ul style="list-style-type: none">• An entity type• Entity matcher class (EntityMatcher subclass) that knows how to compare two entity instances of that type. This mapping behavior is critical to how the comparison between the existing and new policies is performed. The entity matchers determine whether an entity is classified as added, removed, or matched in the policy comparison code. When you add new entities to the policy data model, typically you need to add a corresponding entity matcher class for that entity. If no EntityMatcher is defined for a given type, then ClaimCenter uses an

#	Step Summary	Description	Implementation
			EntityMatcher defined for the nearest supertype. If you add a new entity subtype to the data model that defines no new identifying fields, you do not need to create a new entity matcher for the subtype.
5	Relinking	ClaimCenter finds foreign key links between the claim and policy that would be broken if the new policy replaced the existing policy.	Foreign keys may link from non-policy entities to the policy entities or from policy entities to non-policy entities. This step happens in the policy refresh plugin method called compare. In the base configuration, the matching process is coordinated by the PolicyRefreshConfiguration implementation class. The policy configuration class knows how to create the relink handler objects that perform the relinking.
6	Select risk units	For commercial auto and commercial property lines, ClaimCenter displays a user interface for selecting risk units in the new policy. For example, you can select which insured vehicles or properties are relevant to this particular claim.	The user can check one or more risk items in a list for this commercial line of business. For example, for vehicles ClaimCenter displays one row for each vehicle with a checkbox for each item to select it. Each vehicle row includes the vehicle identification number (VIN), vehicle make, and vehicle model.
7	Display differences between old and new policy	ClaimCenter displays differences between the two policies to the user, with appropriate errors and warnings.	The DiffDisplay interface is the mechanism for configuring: <ul style="list-style-type: none"> • How the application displays differences between objects in the user interface. • How the application displays WARNING, ERROR, or INFO messages as a result of any differences. Each entity can have a corresponding DiffDisplay object to define unique display, warnings, or errors. If no difference item is present for some entity type, ClaimCenter uses the default behavior defined in EntityDiffDisplayBase (and no messages are generated).
8	Relink the claim and If the user approves the policy the policy graphs	refresh, ClaimCenter attempts to relink broken links between Claim-related entities and Policy-related entities. In other words, determine the links between the claim and the old policy and reproduce those links between the claim and the new policy.	This step happens in the policy refresh plugin method called relink. In the base configuration, the policy refresh plugin base plugin uses a class called PolicyRelinker.
9	Remove the old policy graph	ClaimCenter deletes all objects on the old policy. Note that ClaimCenter deletes, not retires, the objects.	This step happens in the policy refresh plugin method called removeOldPolicy.
10	Commit all changes	ClaimCenter commits the bundle. This persists all related database changes in a single transaction.	This step happens in internal code.

Determining the extent of the policy graph

The policy refresh process knows what entities to replace because the objects in the current claim must either belong to the policy graph or not belong to the policy graph. The definition of the extent of the policy graph affects how ClaimCenter compares, relinks, and replaces the policy.

The policy graph within a claim

The formal definition of the policy graph is the set of all entities in the claim graph that the `IPolicySearchAdapter` retrieves from the external policy administration system. Ensure the design of your data model graph cleanly separates the objects that belong to the policy graph from those that do not.

If an object in the claim graph is not part of the policy graph, ClaimCenter treats the object as part of the claim-specific part of the claim graph. Claim objects remain after a policy refresh, although some of their foreign keys that reference objects in the policy graph may change.

If an object in the claim graph is part of the policy graph, policy refresh replaces that object with a newer version, generally speaking. If an object used to exist but is absent on the new policy, for unusual cases you can optionally configure it to remain rather than deleting it.

The policy refresh plugin determines the objects in the policy graph

The policy refresh plugin determines the exact set of policy graph objects using one of the following strategies.

- XSD/WSDL approach – Programmatically examining the remote service that implements the policy retrieval plugin (`IPolicySearchAdapter`) and interpreting the types returned. For example, examine the XSD types in the WSDL that defines the web service connection to the external policy administration system.
- Manual approach – Writing code that traverse the claim and finds all known policy-related entity types in the graph and return the set of objects of those types.

In the base configuration, ClaimCenter provides examples of both approaches.

- The XSD/WSDL approach exists in the policy refresh plugin that is specific to PolicyCenter, `PCPolicyRefreshPlugin`. The policy graph can be automatically determined from the implementation of the `IPolicySearchAdapter`. The `PCPolicyRefreshPlugin` class examines XSD types from the WSDL file called `pctegrationVERSION.wsdl`. This code is implemented in Java and is not visible in Studio.
- The manual approach exists in the `BasicPolicyRefreshPlugin` class, which calls out to the `BasicPolicyRefreshConfiguration` that uses code like that shown below.

```

/*
 * This method extracts all Policy-only entities from the existing Policy (which is linked
 * to a Claim and other non-Policy entities). Note that this implementation uses the getEntireArray()
 * method, which returns all members of the array including retired beans. Including retired beans is
 * necessary here because of issues purging policy entities if retired beans have a foreign key to
 * one of those entities.
 */
override function getPolicyOnly(existingPolicy : Policy) : Set<KeyableBean> {
    var policyOnly = new NonNullSet<KeyableBean>()
    policyOnly.add(existingPolicy)
    getEntireArray(existingPolicy, "Roles", ClaimContactRole).each(\ r -> {
        policyOnly.add(r)
        policyOnly.add(r.ClaimContact)
        includeContact(policyOnly, r.ClaimContact.Contact)
    })
    getEntireArray(existingPolicy, "ClassCodes", ClassCode).each(\ c -> policyOnly.add(c))
    getEntireArray(existingPolicy, "StatCodes", StatCode).each(\ s -> policyOnly.add(s))
    getEntireArray(existingPolicy, "Endorsements", Endorsement).each(\ e -> policyOnly.add(e))
    getEntireArray(existingPolicy, "RiskUnits", RiskUnit).each(\ ru -> includeRiskUnit(policyOnly, ru))
    getEntireArray(existingPolicy, "Coverages", Coverage).each(\ cov -> includeCoverage(policyOnly, cov))
    getEntireArray(existingPolicy, "PolicyLocations", PolicyLocation).each(\ p ->
        includeLocation(policyOnly, p))
    return policyOnly
}

```

Some of the utility functions that this code calls are necessary to query the database for results that include retired objects. It is necessary to find the retired instances in order to support proper purging of old policy objects.

Extending the data model for the policy graph

If your policy refresh plugin uses the manual approach you extend the policy graph in the policy administration system and in ClaimCenter, you must perform additional configuration steps. ClaimCenter calls the policy refresh plugin for to determine the extent of the policy graph within the claim graph. You must ensure your plugin gives the correct answer.

For example, if you base your implementation on `BasicPolicyRefreshPlugin`, you must modify the `getPolicyOnly` method. Modify the method so it includes any instances of your extended entity types that belong to the policy graph in the returned set. If you do not modify the method, ClaimCenter considers your new entity types part of the claim-specific part of the claim graph.

Policy refresh entity matcher details

To replace the existing policy with the new policy and display any differences to the end user, ClaimCenter compares the two policies. This comparison determines the following behaviors.

- Identifies which entities were removed from the current policy
- Identifies which entities were added in the new policy
- Identifies which entities are present in both and may be unmodified or updated. The ClaimCenter terminology for an object existing in both the old policy and the new policy is that the objects match.

In the default implementation of policy refresh, the entity matching happens in classes called entity matcher classes. Entity matcher classes determine which entities in the existing and new policies are logically equivalent. Entity matchers all implement the interface `gw.api.bean.compare.EntityMatcher`. This interface defines a single method: `doEntitiesMatch`, which takes two instances of the type. It returns `true` if and only if the two entities being compared are logically equivalent.

Use of entity matcher classes is the only way to determine whether two entities are equal for policy refresh. In the context of policy refresh, entity matching is not the same as the `Object.equals` contract or the Gosu operators `==` nor `==>`. Generally speaking, two entities are logically equivalent if they are identical on the set of identifying properties, which is usually a subset of all the properties on an object.

Typically, identifying properties are properties on an object itself. Sometimes, an identifying property is a relation to another. For example, two coverage terms match if and only if they relate to a matching coverage.

Some policy objects contain a policy object Policy System ID (PSID) in the `PolicySystemId` property. The PSID is a unique ID that your external policy administration system specifies as uniquely representing the object in the policy administration system. For objects that have a PSID, populate the property so you can match object instances. Two policy objects of the same entity type with matching PSID values definitively match. The policy administration system is responsible for ensuring that the PSID is both unique and immutable.

If you add new entity types, you must write a new matching class for each new entity type, even if you use a PSID to match that entity type.

If you write a new matcher implementation, you must provide a default (no-argument) constructor.

Matching related objects

If an entity matcher must reference other related entities, it also implements the initializable matcher interface `gw.api.bean.compare.InitializableMatcher`. The entity matcher retrieves those entity instances through a matcher context object, which is an instance of `MatcherContext`.

If an entity matcher needs to reference other related entities:

- The entity matcher must also implement interface `gw.api.bean.compare.InitializableMatcher`.
- The entity matcher must retrieve the related entity instances through `MatcherContext`.

The `InitializableMatcher` interface defines the `init` method.

```
public void init(MatcherContext context)
```

If a given matcher implements this interface, the matcher class can implement this method to get (and save in a instance variable) an instance of `gw.api.bean.compare.MatcherContext` to use during matching. Use this object to find an `EntityMatcher` for any given entity type when comparing entities related to the two entity instances passed into `gw.api.bean.compare.EntityMatcher.doEntitiesMatch(a, b)`. To make this easy to use, there is an abstract base class called `gw.api.bean.compare.InitializableMatcherBase` that defines the `doRelatedEntitiesMatch` method. This method compares two related objects and returns true if and only if they match. The method performs the following operations.

1. Finds matcher for the types passed in as arguments
2. Uses that matcher to match the entity by calling the matcher's `doEntitiesMatch` method.

Registering matcher classes

Entity matchers are registered in a `PolicyRefreshConfiguration` implementation in its getter function for the `MatcherTypes` property. The `MatcherTypes` property returns a `java.util.Map` that maps entity types to their entity matcher classes. In the base configuration, the `PolicyRefreshConfigurationBase` class defines the mapping for all built-in existing matchers. For example, this defines the mapping from `Address` to `AddressMatcher`. ClaimCenter uses this map to match entities between the current policy and the new policy.

The `MatcherTypes` property getter in the configuration object looks like the following code.

```
/*
 * Returns a map of type-->EntityMatcher type that is used to match entities between
 * the old and new policies.
 */
override property get MatcherTypes() : Map<IEntityType, Class<EntityMatcher<KeyableBean>>> {
    return {
        Address -> AddressMatcher,
        Building -> BuildingMatcher,
        Contact -> ContactMatcher,
        ...
    }
}
```

In general, if you add a new entity to the policy data model, for any entity that `IPolicySearchAdapter` can retrieve, you must also add an `entityMatcher` class for that entity. You must then map that entity in the `MatcherTypes` getter mapping logic.

If ClaimCenter cannot find an `entityMatcher` class for a given type, it uses the `entityMatcher` defined for the nearest supertype. Therefore, you do not need to define an `entityMatcher` class for an entity that you add as a new entity subtype if all of the following are true.

- The new entity subtype does not define any identifying fields.
- An appropriate matcher class exists for a supertype of the entity.

If you write new entity matcher classes, remember to register them in your `PolicyRefreshConfiguration` implementation in the getter function for the `MatcherTypes` property.

Best practices for entity matcher classes

Guidewire recommends the following for writing matcher classes.

1. Search for a definitive match if you are writing a custom entity matcher.

Definitive matches are not always possible. If a definitive match is not possible through the `AddressBookUID`, `PolicySystemID`, or other identifier, your entity matcher class must attempt to match on fallback criteria. For this reason, Guidewire recommends that you always populate `PolicySystemID` for objects that have this field to make definitive matching easier. Some types do not have that property, typically because there is another unique immutable field, such as the contact `AddressBookUID` property.

2. Match only on fields in an entity that uniquely identify instances of that entity.

You can also test associated (linked) entities in the case in which those identify the given entity. For example, the `PolicyLocation` associated with a `LocationBasedRU`. If this is not possible, you can write fallback logic that

finds probable matches and prioritize the closest match. This is the task of a prioritizer class, which an entity matcher returns from its `getMatchPrioritizer` method. You can also use this class to sort by other criteria in the case in which multiple matches are found. Refer to the API Reference documentation for details of a prioritizer.

In the base configuration, ClaimCenter policy refresh uses its own internal algorithm to determine the closest match if it finds multiple matches. However, ClaimCenter uses the prioritizer if it can determine no other way to narrow down the closest match.

Entity matcher classes in the base configuration

In the base configuration, the `ContactMatcher` class provides default matching for all `Contact` entities. It is overridden for both the `Person` and `Company` entities by the `PersonMatcher` and `CompanyMatcher` matcher implementations respectively. In the base configuration, the `ContactMatcher` class has logic similar to that implemented in the code block below.

```
/*
 * Matches on Address Book UID, then TaxID, then display name.
 */
override function doEntitiesMatch(c1:Contact, c2:Contact) : boolean {

    if(areBothNotNull(c1.AddressBookUID, c2.AddressBookUID)) {
        return c1.AddressBookUID.equals(c2.AddressBookUID)
    }

    if (areBothNotNull(c1.PolicySystemId, c2.PolicySystemId)) {
        return c1.PolicySystemId == c2.PolicySystemId
    }

    if(not c1.Subtype.equals(c2.Subtype)) {
        return false
    }

    if(areBothNotNull(c1.TaxID, c2.TaxID)) {
        return c1.TaxID.equals(c2.TaxID)
    }

    // Attempt to match on display name if none of the above worked.
    return c1.DisplayName.equals(c2.DisplayName)
}
```

Some comments about the code block are discussed below.

- As `ContactManager` is the final authority for contact information, ClaimCenter considers property `AddressBookUID` a definitive match if the same value is present on both entities.
- Failing that, the `PolicySystemId` is a definitive match if present and both values match.
- In the same manner, ClaimCenter uses the same process for other identifiers such as the Tax ID and the DUNS ID.
- If none of the previous IDs match, the last test is a non-definitive one. ClaimCenter checks to see if the company name matches. This is the fallback logic. It is highly unlikely that two or more companies with the same name actually appear on the same policy. However, if this does happen, ClaimCenter will determine the closest match based on other criteria. For example, if the two different companies had two different primary addresses, the Policy Refresh logic would see this and match appropriately.

Other custom matcher logic in the base configuration

The following list describes other examples of matcher logic in the ClaimCenter base configuration.

Address	This matcher logic is similar to the <code>CompanyMatcher</code> . The <code>AddressMatcher</code> logic attempts to definitively match the <code>AddressBookUID</code> if it is present on both entities. Failing this, it uses fallback logic to find a probable match based on the <code>AddressLines</code> , <code>City</code> , <code>County</code> , <code>State</code> , <code>Country</code> , and <code>PostalCode</code> .
RUCoverage	The <code>RUCoverageMatcher</code> attempts to definitively match the <code>PolicySystemId</code> , then matches on the associated <code>RiskUnit</code> .
PolicyLocation	The <code>PolicyLocationMatcher</code> attempts to definitively match the <code>PolicySystemId</code> , then matches on the associated addresses (if both are non-null).

Example of changing matching criteria

You might need to change the matching criteria for some entities. This topic shows how you might change the matching criteria.

The default `VehicleMatcher` logic follows these steps.

- If the `PolicySystemID` values match, the `Vehicle` objects match.
- If the VIN numbers match, the `Vehicle` objects match.
- If the `SerialNumber` values match, the `Vehicle` objects match.
- If both the `LicensePlate` and the `State` (the registered state for the vehicle) match, the `Vehicle` objects match.
- If none of the previous fields match, the `Vehicle` objects do not match.

The code for this default logic is listed below.

```
class VehicleMatcher extends MatcherBase<Vehicle> {

    /*
     * Match on identifying information, or a combination of unique
     * characteristics.
     */
    override function doEntitiesMatch(v1 : Vehicle, v2 : Vehicle) : boolean {

        // Do policy system IDs match?
        if (areBothNotNull(v1.PolicySystemId, v2.PolicySystemId)) {
            return v1.PolicySystemId == v2.PolicySystemId
        }

        // Do vehicle identification numbers match?
        if (areBothNotNull(v1.Vin, v2.Vin)) {
            return v1.Vin==v2.Vin
        }
        // Do serial numbers match?
        else if (areBothNotNull(v1.SerialNumber, v2.SerialNumber)) {
            return v1.SerialNumber==v2.SerialNumber
        }

        // Do license plates match?
        else if (areBothNotNull(v1.LicensePlate, v2.LicensePlate)
                and areBothNotNull(v1.State, v2.State)) {
            return v1.LicensePlate==v2.LicensePlate and v1.State==v2.State
        }

        return false // No identifying properties exist on both objects.
    }
}
```

To alter this behavior so that a match can also occur on the `Make` and `Model` properties of the `Vehicle` objects, add an additional branch to the if/else statement.

```
class VehicleMatcher extends MatcherBase<Vehicle> {

    /*
     * Match on identifying information, or a combination of unique
     * characteristics.
     */
    override function doEntitiesMatch(v1 : Vehicle, v2 : Vehicle) : boolean {

        // Do policy system IDs match?
        if (areBothNotNull(v1.PolicySystemId, v2.PolicySystemId)) {
            return v1.PolicySystemId == v2.PolicySystemId
        }

        // Do vehicle identification numbers match?
        if (areBothNotNull(v1.PolicySystemId, v2.PolicySystemId)) {
            return v1.PolicySystemId == v2.PolicySystemId
        }
        if (areBothNotNull(v1.Vin, v2.Vin)) {
            return v1.Vin==v2.Vin
        }
        // Do serial numbers match?
        else if (areBothNotNull(v1.SerialNumber, v2.SerialNumber)) {
            return v1.SerialNumber==v2.SerialNumber
        }

        // Do license plates match?
        else if (areBothNotNull(v1.LicensePlate, v2.LicensePlate)
                and areBothNotNull(v1.State, v2.State)) {
            return v1.LicensePlate==v2.LicensePlate and v1.State==v2.State
        }

        return false // No identifying properties exist on both objects.
    }
}
```

```

    }
    // Added condition - Do vehicle make and model match?
    else if(areBothNotNull(v1.Make, v2.Make) and areBothNotNull(v1.Model, v2.Model) {
        return v1.Make==v2.Make and v1.Model==v2.Model
    }

    return false // No identifying properties exist on both objects.
}
}

```

Policy refresh relinking details

Relinking is the process of repairing broken foreign keys from a claim entity to a policy entity or from a policy entity to a claim entity. It is important that ClaimCenter repairs any links that potentially are broken whenever replacing the old policy graph with a new policy graph.

For example, if an **Incident** on a **Claim** refers to a **RiskUnit** on the **Policy**, then the claim incident foreign key to the policy risk unit potentially must be repaired. The default behavior is described below.

- If the old and new risk units differ, ClaimCenter sets the foreign key on the incident to the new risk unit on the refreshed policy.
- If the old risk unit does not exist on the refreshed policy, ClaimCenter sets the foreign key on the incident to null.

ClaimCenter encapsulates critical parts of relinking through objects called relink handlers. Relink handlers define the behavior when foreign keys between claim objects and policy objects break during policy refresh. The default behavior depends on the direction of the broken link, as the following table describes.

Direction of link	Relink handler behavior
Claim to policy	If there is a match, move target of link to new policy object.
	If no match exists and the foreign key property in the data model is set to <code>nullok="true"</code> , set the foreign key to null. Otherwise, throw an exception.
Policy to claim	If there is a match, move source of link to new policy-specific object. Otherwise, do nothing because soon ClaimCenter will delete the old policy object.

You can customize this behavior by changing an existing custom relink handler or by adding a new one that implements the interface `gw.api.policy.refresh.relink.RelinkHandler`. The `RelinkHandler` interface defines relinking behavior for all foreign key fields on that entity type.

As a convenience, there is another interface called `gw.api.policy.refresh.relink.LinkHandler` that defines more granular per-link (for each foreign key field) behavior. To use this more granular behavior, define the `RelinkHandler` implementations that extend `PerLinkHandler`. Next, use the `register` method to define the relink behavior on each `LinkHandler` that you want to define.

The default behavior for relinking is typically satisfactory for most entity types and does not require new relinking handler classes. However, you might need to customize relink handlers if you want to handle a broken link in a special way. For example, suppose a link is to an object that no longer exists in the new policy. In the default case, that would cause an error. In special cases, you might want to define some more complex compensation behavior. Contact Customer Support if you think you need to modify behavior of links broken during policy refresh.

Configure any new relink handlers in the `PolicyRefreshConfiguration` property getter called `CustomRelinkerTypes`. See the `PolicyRefreshConfigurationBase` class for the default mapping.

Some examples of built-in custom relink handling are listed below.

- Relinking `ContactTag` objects to a new policy contact.
- Unlinking the `Deductible` entity from financial transactions when removing the corresponding `Coverage` object.

Re-link handlers must be defined for the specific entity type that owns the link. This means that you cannot specify relink handlers as shared code in the supertype for all its subtypes. This is notable because in contrast, entity matchers for new subtypes can share implementation in their supertype.

Example relink handler class for contact tag objects

In the base configuration, ClaimCenter provides the following example of a custom relink handler class. The `ContactTagRelinkHandler` class relinks `ContactTag` objects to the new policy `Contact`.

```
class ContactTagRelinkHandler extends PerLinkHandler<ContactTag> {

    construct() { register(ContactTag, "Contact", new TagLinkHandler()) }

    /*
     * LinkHandler for ContactTag-->(Contact)-->Contact
     */
    private class TagLinkHandler extends BaseLinkHandler<ContactTag> {
        /*
         * Handle the case where the Contact is in the removed set.
         */
        override function handleRemovedLinkTarget(relinkItem:RelinkItem<ContactTag>,
            relinkCtx:RelinkContext, target:KeyableBean) {
            // Ignore if not matched.
        }

        /*
         * Handle the case where the new Contact already has the ContactTag being relinked.
         */
        override function handleClaimToPolicyLink(relinkItem:RelinkItem<ContactTag>,
            relinkContext:RelinkContext) {
            var target = relinkItem.Value as Contact

            if (relinkContext.PolicyComparison.hasMatch(target)) {
                var newBean = relinkContext.PolicyComparison.findMatch(target).Compare

                if (not hasContactTag(relinkItem.Owner, newBean)) {
                    relinkItem.setValue(newBean)
                } else {
                    // Do nothing if no match.
                    handleRemovedLinkTarget(relinkItem, relinkContext, target)
                }
            }
        }

        private function hasContactTag(tag:ContactTag, contact:Contact) : boolean {
            return contact.Tags.firstWhere(\ t -> t.Type==tag.Type) != null
        }
    }
}
```

Relink filters

ClaimCenter uses relink filters to define special logic that is outside the default policy refresh behavior. Relink filters are only for logic that needs to execute before or after relinking. ClaimCenter executes relink filters before the relinking step (in a `preProcess` method) and after the relinking step (in a `postProcess` method). Relink filters implement the interface `gw.api.policy.refresh.relink.RelinkFilter`.

Examples of default relink filters are described in the following table.

Relink filter class	Purpose
<code>ContactPreservingFilter</code>	Moves any removed Contacts and moves them to roles with the added prefix <code>former_</code> .
<code>ClaimLinkPreservingFilter</code>	Preserves any policy objects if they are removed and referenced from claim entities.
<code>CopyInternalFieldsFilter</code>	Copies internal fields from policy entities that are matched to their corresponding replacements in the new policy.

Best practices for writing relink filters

Guidewire recommends the following practices for writing relink filters.

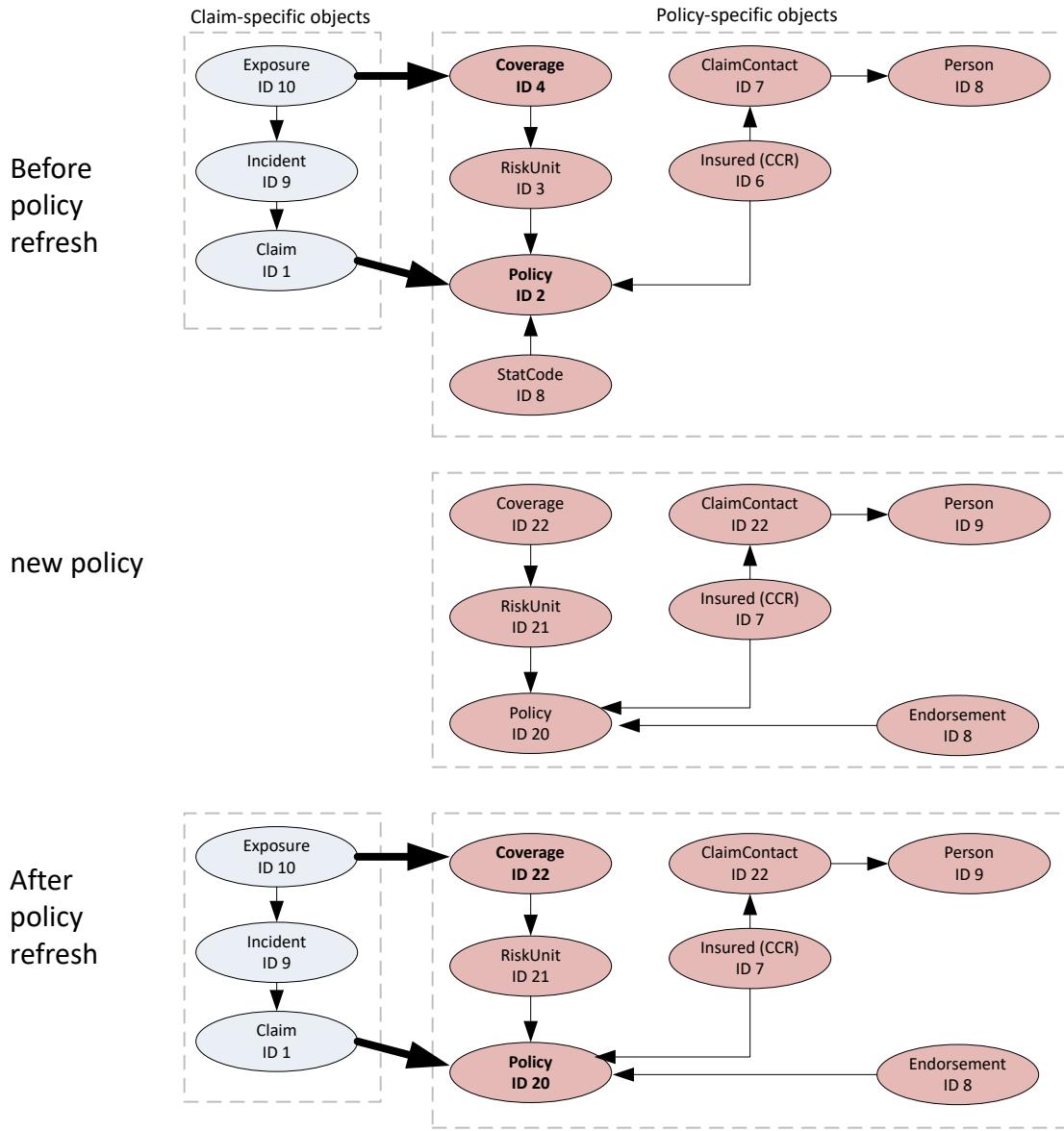
1. Relink filters are only for logic that needs to execute before or after relinking.
2. Use the `ClaimLinkPreservingFilter` to preserve entities instead of trying to preserve entities directly from the `gw.api.policy.refresh.relink.RelinkContext.addExcludedFromRemoval` in a `RelinkHandler` class. Attempting to preserve entities from the handler can lead to subtle order dependencies in how ClaimCenter performs relinking.

3. Limit preservation behavior to entities that have no outgoing foreign keys (such as `Address`) and use preservation sparingly.
4. If you need to handle only a few foreign keys differently, refer to earlier in the topic about the link handler called `PerLinkHandler`. In that case, do not use a relink filter.
5. As with all relinking code, be mindful of how the relinking can potentially interact with other ClaimCenter processes such as Preupdate rules or bundle callbacks.

Example: Policy refresh matching and relinking with relink set information

Let us consider the earlier example of relinking and removal. The following diagram shows the state of the claim-policy graphs both before and after policy refresh.

Policy Refresh and Relinking



→ A regular foreign key link that does not cross the claim-policy boundary

→ Foreign Key link between a claim-specific object and a policy-specific objects. These are the links that will break when replacing the policy. It is these links that must be relinked.
 Note: this is a simplified diagram. In a real-world system, there would also be links that point from the policy-specific objects to one of the claim-specific objects. ClaimCenter also relinks those links during policy refresh.

Notice the following differences between the old and new policy.

- The new policy no longer has the StatCode.
- The new policy has a new entity, an Endorsement.
- The claim graph's Exposure with ID=10 now points to the new Coverage.

- The claim graph's `Claim` with ID=1 now points to the new `Policy`.

After getting the new policy graph, ClaimCenter traverses the new policy. Next, ClaimCenter invokes matcher classes on entities of the same type and in the same graph position in the two graphs.

- The following entities are logically the same and the matcher classes add these to the matched set: `Policy`, `RiskUnit`, `Coverage`, `ClaimContactRole`, `ClaimContact`, `Person`.
- The `StatCode` object is absent in the new policy and thus becomes part of the set of removed objects.
- The `Endorsement` from the new policy object is absent in the old policy and thus becomes part of the set of added objects.

After the policy comparison, ClaimCenter must find all links between the group of policy objects and the group of non-policy claim objects. In this simplified example, the links are **Claim → Policy and Exposure → Coverage**. These are the foreign key references that ClaimCenter must relink as part of completing policy refresh. In a real-world example, there might also be links from policy objects to claim objects.

Policy refresh policy comparison display

ClaimCenter displays a side-by-side comparison of the current and new policy versions in the **Policy Comparison** screen at policy refresh. Within this screen, it is possible to configure the following components.

- The objects and properties that ClaimCenter shows in the policy comparison tree.
- The order, labels, and values for objects and properties that ClaimCenter shows in the **Policy Comparison** screen.
- The warning and error messages that ClaimCenter generates if it detects an issue during policy refresh.

The **Policy Comparison** screen uses two main types of objects to configure differences and display the differences.

- Difference objects – These represent a match between an entity from the current policy graph and new policy graph, or any detected property differences.
- Difference display objects – These display a diff object, and can generate errors and warnings.

Difference objects

A difference object (a `Diff` object, or a `diff`) represents a match between an entity from both current and new policy graphs, or a difference in a property. There are two types of difference objects: entity difference objects and property difference objects.

Entity difference objects

An entity difference object has a name like `entityDiff`, for example `PolicyDiff`. ClaimCenter creates an entity difference object in the policy comparison tree in the following situations.

- The object has been added or removed from the policy.
- The object has child objects that are changed, added, or removed.
- The object has properties that changed.

A logical pair of entities consists of either of the following entities.

- One entity from each policy graph that the matcher classes matched.
- One entity from only one policy graph, which means that an entity was not matched and the new policy either added or removed the entity from the policy.

The foreign key properties are described in the following table.

Difference object property	Meaning of value
SourceValue	If the current entity exists, contains a foreign key to the entity. If an element of a policy is being added, then <code>Diff.SourceValue</code> is null. This means that the entity appears only in the new policy graph.
CompareValue	If the new entity exists, contains a foreign key to the entity. If an element of a policy is being removed, then <code>Diff.CompareValue</code> is null. This means that the entity appears only in the current policy graph.

Property difference objects

ClaimCenter creates a property difference (`PropertyDiff`) if an important policy property changed, such as one of the following properties.

- `CancellationDate`
- `PolicySystemPeriodID`
- `ProducerCode`

Entity difference objects

An entity difference object has a name like `entityDiff`, for example `PolicyDiff`. ClaimCenter creates an entity difference object in the policy comparison tree in the following situations.

- The object has been added or removed from the policy.
- The object has child objects that are changed, added, or removed.
- The object has properties that changed.

A logical pair of entities consists of either of the following entities.

- One entity from each policy graph that the matcher classes matched.
- One entity from only one policy graph, which means that an entity was not matched and the new policy either added or removed the entity from the policy.

The foreign key properties are described in the following table.

Difference object property	Meaning of value
SourceValue	If the current entity exists, contains a foreign key to the entity. If an element of a policy is being added, then <code>Diff.SourceValue</code> is null. This means that the entity appears only in the new policy graph.
CompareValue	If the new entity exists, contains a foreign key to the entity. If an element of a policy is being removed, then <code>Diff.CompareValue</code> is null. This means that the entity appears only in the current policy graph.

Property difference objects

ClaimCenter creates a property difference (`PropertyDiff`) if an important policy property changed, such as one of the following properties.

- `CancellationDate`
- `PolicySystemPeriodID`
- `ProducerCode`

Difference display objects

At a programming level, the main mechanism for configuring both how ClaimCenter displays differences on the policy refresh Policy Comparison screen is the `DiffDisplay` interface. Each entity type has a corresponding object that

implements `DiffDisplay` to display differences for objects of that type. Objects that implement this interface are called difference display objects, or `DiffDisplay`, objects.

Difference display objects are also the mechanism for generating warnings, errors, or informational messages as a result of any differences. Any errors actually block policy refresh from proceeding.

There are different types of `entityDiffDisplay` classes that display entity-level information for each type of entity in the policy graph, and a single `PropertyDiffDisplay` class. The following table summarizes the differences.

Class	Purpose
<code>entityDiffDisplay</code>	<p>Each entity difference display class (an object with name <code>entityDiffDisplay</code>) defines how to display difference information for that entity type as well as what refresh messages to display, if any. Every <code>entityDiffDisplay</code> class extends <code>EntityDiffDisplayBase<T></code>.</p> <p>For any entity type that appears in the policy graph but for which there is no specific <code>entityDiffDisplay</code> class, ClaimCenter uses the entity difference display base class called <code>EntityDiffDisplayBase</code>. In practice, for the defaults ClaimCenter relies on <code>KeyableBeanDiffDisplay</code>, which extends <code>EntityDiffDisplayBase</code>.</p> <p>In the default configuration, ClaimCenter defines custom difference display objects for the following entity types: <code>Policy</code>, <code>RUCoverage</code>, <code>PolicyCoverage</code>, <code>ClassCode</code>, and <code>PropertyItem</code>.</p>
<code>PropertyDiffDisplay</code>	<p>The property difference display class displays property-level information for all properties. This class identifies how to display difference information for properties, regardless of the entity to which the property belongs. There is only one class with this name. Unlike the naming pattern of entity difference display classes, property difference display objects are not multiple variants of the name for each entity type or property name.</p> <p>The class <code>gw.plugin.policy.refresh.ui.PropertyDiffDisplay</code> class is an internal Java class that you cannot modify.</p>

Add difference display object or modify mapping

To display differences between entity instances in a policy graph, you use an `entityDiffDisplay` object for that entity type.

Procedure

1. If an `entityDiffDisplay` class exists for that entity, modify that existing `entityDiffDisplay` class.
2. If no `entityDiffDisplay` class exists for that entity, do both of the following operations.
 - a) Create a new `entityDiffDisplay` class that extends `EntityDiffDisplayBase<T>`.
 - b) Add an appropriate map element to the `DiffDisplayTypes` getter to the `PolicyRefreshConfiguration` object that you are using, or modify the built-in `PolicyRefreshConfigurationBase` class.

Example

To see the mapping in the base configuration, see the `PolicyRefreshConfigurationBase` Gosu class in the `DiffDisplayTypes` property getter.

In the base configuration, ClaimCenter defines `entityDiffDisplay` classes for the following entities.

- `Policy`
- `Coverage`
- `PolicyCoverage`
- `RUCoverage`
- `ClassCode`
- `PolicyLocation`
- `PropertyItem`

The default configuration contains the following example code for the getter.

```
override property get DiffDisplayTypes() : Map<IEntityType, Class<DiffDisplay>> {
    return { ClassCode -> ClassCodeDiffDisplay,
             KeyableBean ->KeyableBeanDiffDisplay,
             Policy -> PolicyDiffDisplay,
             PolicyCoverage -> PolicyCoverageDiffDisplay,
             PropertyItem -> PropertyItemDiffDisplay,
             RUCoverage -> RUCoverageDiffDisplay
    }
}
```

If the map does not list a specific policy graph entity type, ClaimCenter uses the `KeyableBeanDiffDisplay` class as the default, which extends `EntityDiffDisplayBase`.

Getting the difference object from an entity difference display object

Every `entityDiffDisplay` class inherits a `Diff` property from the `EntityDiffDisplayBase` class that it extends. Use the `Diff` property to access the related `entityDiff` instance.

- Use `Diff.SourceValue` to access fields on the current policy.
- Use `Diff.CompareValue` to access fields on the new policy.

For example, the following code determines if the effective date on the new policy differs from the effective date on the current policy by more than seven days. If so, the code generates an error message to that effect.

```
if (Diff.CompareValue.EffectiveDate != Diff.SourceValue.EffectiveDate) {
    if (Diff.SourceValue.EffectiveDate.daysBetween(Diff.CompareValue.EffectiveDate) > 7 ) {
        messages.add(UIMessage.error("The effective date has changed by more than 7 days."))
    }
}
```

Getting the type of difference from an entity difference display object

Every `entityDiffDisplay` class inherits a `Type` property from the `EntityDiffDisplayBase` class that it extends. This property specifies the type of difference involved between the current and new entity instances. The different values of `Type` have the following meanings.

Type property value	Meaning
ADDED	The entity instance appears only in the new policy.
REMOVED	The entity instance appears only in the existing or current policy.
CHANGED	The entity instance appears in both policies, but its properties and/or child entities are not the same.
UNCHANGED	The entity instance appears in both policies and its properties and/or child entities are the same.
MOVED_TO	The entity instance appears in both the new and the existing policies. However, it does not have the same parent in each policy.
MOVED_FROM	This situation can occur if you move a single child from one parent to another. For example, suppose a business has several garage locations and several vehicles. Between two versions of the policy, it is possible for a single vehicle to appear garaged in one location on the current policy. It can then appear in a different location in the new policy. The entity thus appears twice in the comparison tree. <ul style="list-style-type: none"> • The entity will have the <code>MOVED_FROM</code> type status on the old parent and will appear in the comparison tree using the <code>REMOVED</code> icon.

Type property value	Meaning
	<ul style="list-style-type: none"> The entity will also have the MOVED_TO type status on the new parent and will appear in the comparison tree using the ADDED icon.

For example, the following code from `PolicyDiffDisplay` looks at the current and new versions of the policy and determines if the currency changed on the policy. If so, it generates either an error or a warning, depending on whether the currency on the policy is editable. In general, a policy is editable if the claim associated with this policy does not have any transactions.

```
if (Type==CHANGED) {
    if (Diff.SourceValue.Currency!=Diff.CompareValue.Currency) {
        if (not Diff.SourceValue.CurrencyEditable) {
            messages.add(UIMessage.error(displaykey.PolicyRefresh.DiffDisplay.Policy.CurrencyChange))
        } else {
            messages.add(UIMessage.warning(displaykey.PolicyRefresh.DiffDisplay.Policy.CurrencyChange))
        }
    }
}
```

Policy refresh errors, warnings, and informational messages

Each `DiffDisplay` object is responsible for generating any messages for that entity type by returning a list of messages in its `getMessages` method. Policy refresh messages include error, warning, or information messages. ClaimCenter encapsulates each message in an object of type `gw.api.web.UIMessage`.

Generally speaking, a difference display object makes messages that are related only to the entity for which the given `DiffDisplay` object is responsible.

- An **ERROR** message is the only message type that actually blocks the policy refresh from completing. Define these for incompatible changes that cannot be resolved automatically using existing mechanisms. Avoid throwing exceptions from within the policy refresh process unless you cannot represent them at the user interface level by an appropriate **ERROR** message.

If ClaimCenter identifies an error condition, it removes the **Finish** button from the **Policy Comparison** screen. This prevents the policy refresh from completing.

- WARNING** messages define issues that might pose a problem, but are not automatically conditions that block policy refresh.
- INFO** messages define any other informative message or suggested action as a result of the policy refresh. In the base configuration, there are no information messages generated for built-in entity types.

ClaimCenter lists all messages in order of severity. In other words, ClaimCenter lists all errors first, followed by warnings, and then by info messages.

You can configure all message types. In configuring messages, the following operations are performed.

- Construct each policy refresh message as an instance of `gw.api.web.UIMessage`.
- Group a set of `UIMessage` objects through the use of `gw.api.web.UIMessageList`.
- Use the method `entityDiffDisplay.getMessages` to construct the `UIMessageList` object, which ClaimCenter renders in the user interface as the set of error, warning, and informational messages.

Creating a policy refresh message

To create a policy refresh message, a difference display object creates a `UIMessage` object.

Use the following syntax to call static methods on the `UIMessage` class.

```
UIMessage.error(msgText)
UIMessage.warning(msgText)
UIMessage.info(msgText)
```

Always use a display key for the message text.

```
UIMessage.warning(displaykey.PolicyRefresh.DiffDisplay.Policy.PolicySystemPeriodIDChange))
```

In addition to being shown in the policy comparison screen, error messages also block the refresh process. There is no additional functionality associated with warning or info messages.

In the base configuration, there are a few *entityDiffDisplay* classes that do not create messages. One, for example, is the *PolicyCoverageDiffDisplay* class. However, of the classes that do create messages, most of these classes create messages directly in the *getMessages* method. The one exception is the *CoverageDiffDisplay* class, which merely calls a *performValidation* method from within the *getMessages* method. ClaimCenter actually creates the messages within this *performValidation* method, which exists in *gw.plugin.policy.refresh.CoverageValidator*.

Return a policy refresh message list to encapsulate messages

To display messages about differences between entity instances in a policy graph, you use an *entityDiffDisplay* object for that entity type.

About this task

ClaimCenter encapsulates a set of *UIMessage* objects to show to the user in a *UIMessageList* object.

Procedure

1. Create a new *UIMessageList* object using the new operator.

```
messages = new UIMessageList()
```

2. Define the condition to use for comparing the two policy versions.

```
Type==CHANGED
```

3. Compare the new and current entities or properties that are of interest.

```
Diff.SourceValue.Currency!=Diff.CompareValue.Currency
```

4. Construct a *UIMessage* object and add it to the *UIMessageList* object using the list's *add* method.

Example

The following code statement demonstrates the process.

```
var messages = new UIMessageList()
if (Type==CHANGED) {
    if (Diff.SourceValue.Currency!=Diff.CompareValue.Currency) {
        messages.add(UIMessage.error(displaykey.PolicyRefresh.DiffDisplay.Policy.CurrencyChange))
    }
}
```

Policy refresh message considerations

For each entity type that can require a message, you must override the corresponding *entityDiffDisplay.getMessages* method. The *getMessages* method receives a *PolicyRefreshMessageContext* object. The *getMessages* method returns a *UIMessageList* that is either empty or contains one or more messages to show within ClaimCenter.

The *PolicyRefreshMessageContext* object provides context for the difference display object code to construct errors and warning related to a policy refresh. A *PolicyRefreshMessageContext* object has two methods that are useful in policy refresh message logic that each returns the set of non-policy claim entities linked to a given entity.

- The *findClaimEntitiesLinkedTo* method returns all non-policy claim entities.
- The *findClaimEntitiesOfTypeLinkedTo* method filters for entities of a specific type.

For example, the following code appears in the base configuration `PropertyItemDiffDisplay` class.

```
if (this.Type==REMOVED) {
    if (ctx.findClaimEntitiesOfTypeLinkedTo(Diff.SourceValue, PropertyContentsScheduledItem).
        HasElements) {
        result.add(UIMessage.warning(displaykey.PolicyRefresh.DiffDisplay.PropertyItem.
            MissingPropertyItem(Diff.SourceValue)))
    }
}
```

The logic for the example code is described below.

- If this property item is on the current policy, but not on the new policy...
- And, if there are any entities of type `PropertyContentsScheduledItem` on the claim that are linked to the property item on the current version of the policy...
- Then, add the following warning message.

The `PropertyItem` {0} has been removed from the policy and is referenced by the claim

In the display key string, {0} is the display name for the property item.

Configuring visibility in the comparison tree

ClaimCenter uses a RowTree PCF widget to generate the policy comparison tree in the **Policy Comparison** screen. The policy comparison tree displays differences between a current and new policy as a collapsible hierarchy of objects. You can expand or collapse the hierarchy through the use of the +/- controls to expand or collapse a portion of the tree. ClaimCenter shows the +/- controls for entities with child entities only.

It is possible for the leaves of the policy comparison tree (the entries that do not have +/- controls) to be either properties or entities. In the base ClaimCenter application, you can identify whether a leaf is an entity or a property in the following ways.

- The label format for entities is `entityType:DisplayName`, with a colon.
- The label format for properties is `DisplayName`. The ClaimCenter base configuration does not use colons for properties.
- The visual indicator for a changed entity is an icon. The ClaimCenter base configuration shows visual indicators for added or removed entities only.
- The visual indicator for a changed property shows the actual difference, for example, the two different cancellation dates.

Setting visibility

ClaimCenter uses the `entityDiffDisplay.isChildVisible` method to determine what to show in the policy comparison tree. In the base ClaimCenter configuration, Guidewire provides an overridden definition of this method in class `EntityDiffDisplayBase`.

- If you want to globally modify the logic for displaying children, modify the `isChildVisible` method in `EntityDiffDisplayBase`.
- If you want to modify the logic for displaying children for a specific type of entity, then override the `isChildVisible` method in the `entityDiffDisplay` class for that entity. For example, if you want to override the logic for displaying coverages in the policy comparison tree, then you need to modify the `isChildVisible` method in `CoverageDiffDisplay`.

ClaimCenter automatically excludes several sets of properties from the policy comparison tree.

1. ClaimCenter automatically excludes properties used to track an object internally, for example, `ID`, `CreateTime`, and `CreateUser`. This behavior occurs in internal code and you cannot configure it.
2. ClaimCenter programmatically excludes additional properties that you specify through the `PolicyRefreshConfigurationBase.EntityPropertiesToIgnoreForComparison` getter. You configure this

getter to identify additional properties to always omit from the tree. In the base application, ClaimCenter excludes the following properties.

- `Contact.AutoSync`
- `ClaimContact.ClaimantFlag`
- `ClaimContactRole.PartyNumber`

Construction of the policy comparison tree

To construct the policy comparison tree, ClaimCenter performs the following operations.

1. ClaimCenter instantiates an `entityDiff` object for the following items.

- One for every matched pairs of entities
- One for every unmatched entity

ClaimCenter uses logic from the matcher classes to determine if one entity matches another.

2. ClaimCenter organizes the `entityDiff` instances into a tree structure, using the `PolicyRefreshConfigurationBase.DisplayTree` getter.

ClaimCenter uses the `DisplayTree` getter to convert the policy graph into a one-to-many hierarchy for viewing in the ClaimCenter interface. In actuality, the policy data model, and, therefore, the graph of `entityDiff` objects, is not a tree. However, ClaimCenter uses a `RowTree` widget to display the information in the ClaimCenter interface. The `DisplayTree` property transforms the graph of `entityDiff` objects into a tree for viewing purposes.

3. ClaimCenter instantiates an `entityDiffDisplay` object for each `entityDiff` object. The `entityDiffDisplay` object specifies how to display the entity differences.

- Each `entityDiffDisplay` determines whether to display its data, and if so, how to display the data.
- Each `entityDiffDisplay` has a type, which is typically set to ADDED, CHANGED, or REMOVED.
- Each `entityDiffDisplay` is responsible for generating warning or error messages at the top of the **Policy Comparison** screen. There are no messages for `PropertyDiffDisplay`. ClaimCenter only generates messages at the `entityDiffDisplay` level.

Class `PropertyDiffDisplay` exists in the `gw.plugin.policy.refresh.ui` package. It is an internal Java class that you cannot modify.

4. ClaimCenter renders the policy comparison tree based on the logic defined by the `entityDiffDisplay` instances.

Defining the tree using the display tree property getter

The `PolicyRefreshConfigurationBase` class contains the `DisplayTree` property getter. The `DisplayTree` getter builds the policy comparison tree using multiple `addOneToMany` methods to build the tree by connecting child `entityDiff` objects to parent `entityDiff` objects.

The only thing you can configure in the `DisplayTree` getter is the mapping from a given parent to a given child. You cannot create relationships that do not already exist in the policy graph. For example, you cannot display vehicle coverages as the child of a contact.

The `DisplayTree` getter logic uses the following important methods.

`addOneToMany` Adds one-to-many child elements to a given parent element.

`addOneToOne` Adds a single child element to a given parent element.

These methods have three parameters, which you use to specify a parent/child relationship in the policy comparison tree.

`parentType` An `IEntityType` for the parent entity type to which the `entityDiff` object refers.

`propertyName` A String for the property name of the parent property that points to the child.

`childType` An `IEntityType` for the child entity type to which the `entityDiff` object refers.

The `entityDiff` graph consists of a number of nodes (the objects in the graph) and edges (the connections between pairs of related nodes). To construct the tree, ClaimCenter traverses the entire `entityDiff` graph. In doing so, ClaimCenter compares every connection between the tree nodes to the list of relationships specified in the `addOneToMany` and `addOneToOne` methods.

In performing the comparison, the following resulting operations can occur.

- If ClaimCenter finds a relationship in which all parameters match, then it creates an `entityDiffDisplay` instance from the child `entityDiff` and attaches it to the tree at the appropriate point.
- If ClaimCenter does not find a relationship in which all parameters match, then it does not create an `entityDiffDisplay` for that edge or connection.

Guidewire recommends that you pay close attention to how you modify the `DisplayTree` getter, as minor errors can result in ClaimCenter not displaying nodes in the tree. For example, ClaimCenter does not create an `entityDiffDisplay` instance and does not display the information in the policy comparison tree if you do the following actions.

- If you identify the wrong `IEntityType` for the parent or for the child
- If you misspell the name of the relationship

Pay close attention to how you modify the `DisplayTree` getter, because minor errors can result in ClaimCenter not displaying nodes in the tree.

Configuring labels and display order

It is possible to configure the following items in the policy comparison tree.

- Entity and property labels
- Entity and property display order

Configuring entity labels

In the base configuration, ClaimCenter uses the following format for entity labels.

```
entityType:DisplayName
```

Use `entity.EntityName` display keys to format the entity type.

Alternatively, you can configure how ClaimCenter displays an entity label by overriding the `Label` getter in the appropriate `entityDiffDisplay` class. For example, to change how ClaimCenter displays the label for the policy root in the policy comparison tree, add code similar to the following to `PolicyDisplayDiff`.

```
override property get Label() : String {  
    return "Policy (ID: " + Diff.SourceValue.PolicyNumber + ")"  
}
```

Configuring property labels

ClaimCenter generates property labels through an internal lookup search for a display key with the following format.

```
entity.entityType.propertyName
```

Thus, you configure property labels by changing the value of the display key. For example, suppose that you want to change the label **Policy System Period ID** to **Policy Period ID**. You need merely find the correct display key, in this case, `entity.Policy.PolicySystemPeriodID`, and change it to the desired value.

Display key	Value
entity.Policy.PolicySystemPeriodID	Policy System Period ID
entity.Policy.PolicyPeriodID	Policy Period ID

Configuring display order

The policy comparison tree RowTree widget controls property and entity display order. In the base application, the following program behaviors are implemented.

- ClaimCenter lists properties first, in alphabetical order by sort expression.
- ClaimCenter lists entities next, in alphabetical order by sort expression.

A sample display listing is shown below. Notice that properties labels do not contain colons.

```
Policy: 54-847654
Cancellation Date
Policy System Period ID
Producer Code
Covered Vehicle: Honda GX
Person: Allen Robertson
Person: Karen Egertson
```

Within the **Policy Comparison** screen, ClaimCenter labels the left-most column as **Entity or Property** and provides a small triangular sort icon. This column maps to the RowTree widget on the PolicyComparisonScreen PCF. Within the RowTree widget, this column maps to the left-most widget cell, with an ID of **LabelField**. This cell contains the following sorting-related properties.

- **sortBy**
- **sortDirection**
- **sortOrder**

You use the **sortBy** property on the PolicyComparisonScreen PCF to define logic for dynamically generating a sort value for each child element. ClaimCenter then sorts the child elements based on their sort values.

You set the **sortBy** property by adding a **getSortBy** function and defining that function on the **Code** tab. In the base configuration, the **getSortBy** function looks similar to the following code block.

```
uses gw.plugin.policy.refresh.ui.DiffDisplay
uses gw.plugin.policy.refresh.ui.EntityDiffDisplay

function getSortBy (diff : DiffDisplay) : String {
    return ((diff typeis EntityDiffDisplay) ? "B:" : "A:") + diff.Label
}
```

In the base configuration, this function checks to see if the child element is of type **EntityDiffDisplay**.

- If the child element is an **EntityDiffDisplay**, then its sort expression is “B:” plus the label.
- If the child element is not an **EntityDiffDisplay**, then it is a **PropertyDiffDisplay** and its sort expression is “A:” plus the label.

The resulting behavior is described below.

- ClaimCenter sorts all the properties first as they all start with “A:” and then orders the properties by their label value.
- ClaimCenter sorts all the entities afterwards as they all start with “B:” and then orders the entities by their label values.

Be aware that the strings “A:” and “B:” are somewhat arbitrary. It is possible to replace these strings with other strings that sort the properties and entities in the desired way (such as “1”, “2”, “3”...).

To modify the sort order for properties or entities, you need to modify the **getSortBy** function defined on the **Code** tab of the PolicyComparisonScreen PCF. To access the **Code** tab, you need to select the entire PCF.

For example, suppose that you want to configure the sort order so that ClaimCenter displays the Producer Code property before any other property. To do so, modify the `getSortBy` function in the following manner.

```
function getSortBy (diff : DiffDisplay) : String {
    if (diff.Label == "Producer Code") {
        return "A:Producer Code"
    } else {
        return ( (diff typeis EntityDiffDisplay) ? "C:" : "B:") + diff.Label
    }
}
```

Changing icons for added, removed, unchanged, and changed

You can change the icons that ClaimCenter displays for added, removed, unchanged, or changed differences. Change the appropriate display keys that reference file names in the images folder:

- `Web.PolicyRefresh.PolicyComparisonScreen.IconFieldFileName.Added`
- `Web.PolicyRefresh.PolicyComparisonScreen.IconFieldFileName.Removed`
- `Web.PolicyRefresh.PolicyComparisonScreen.IconFieldFileName.Unchanged`
- `Web.PolicyRefresh.PolicyComparisonScreen.IconFieldFileName.Changed`

Policy refresh configuration examples

You can configure policy refresh in multiple ways.

Adding a claim-specific entity to policy refresh

Suppose you add a new entity type, `ClaimEntity_Ext`, that is claim-specific and not part of the policy graph. In `ClaimEntity_Ext` you include a foreign key to `Claim`. In turn, you extend `Claim` with a one-to-one element to provide a symmetrical link back to `ClaimEntity_Ext`.

To ensure `ClaimEntity_Ext` is part of the claim graph, do not modify the policy refresh plugin to include it in the set of entity types that comprise the policy graph. Thus, ClaimCenter knows that your new `ClaimEntity_Ext` entity is part of the claim graph.

Now, suppose your new `ClaimEntity_Ext` has an array property that contains `Vehicle` instances. The `Vehicle` entity is a built-in type that belongs to the policy graph. For array properties, database-level foreign keys must reside on the child objects (`Vehicle`) to relate the children to their parent objects (`ClaimEntity_Ext`). So, you extend `Vehicle` with a foreign key to `ClaimEntity_Ext`.

Example of extending a claim-specific entity type metadata definition

You define your new entity `ClaimEntity_Ext` in the following metadata definition file.

`ClaimCenter/config/extensions/ClaimEntity_Ext.eti`

The contents of the file might look like the following example.

```
<?xml version="1.0"?>
<entity
    xmlns="http://guidewire.com/datamodel"
    entity="ClaimEntity_Ext"
    table="claimentity"
    type="retireable">
    <implementsEntity name="Extractable"/>
    <foreignkey name="Claim" fkentity="Claim"/>
    <array name="Vehicles" arrayentity="Vehicle"/>
</entity>
```

In `ClaimCenter/config/extensions/Vehicle.etx`, extend the `Claim` entity with a one-to-one element.

```
<onetooone fkentity="ClaimEntity_Ext" name="ClaimEntity_Ext" nullok="true"/>
```

In ClaimCenter/config/extensions/Vehicle.etcx, extend the Vehicle entity with a foreign key.

```
<foreignkey name="ClaimEntity_Ext" fkentity="ClaimEntity_Ext"/>
```

The changes described in this example are common types of data model changes. Because your new entity belongs to the claim graph and not the policy graph, the policy refresh system places no additional demands on the configuration.

How policy refresh updates objects in the claim and policy graphs

When policy refresh happens, `ClaimEntity_Ext` entity instances remain unchanged within the claim graph. For links to `ClaimEntity_Ext` instances from refreshed `Vehicle` instances, the policy refresh system relinks them automatically. The directionality of the foreign key runs from a policy object to a claim object. Directionality of foreign keys is an important distinction when reviewing the default relinking behavior of the policy refresh system.

Adding a policy-specific entity to policy refresh

Suppose you add a policy entity type, `PolicyEntity_Ext`, that is policy-specific and not part of the claim graph. In `PolicyEntity_Ext` you include a foreign key to `Policy`. In turn, you extend `Policy` with an array property to provide a collection of symmetrical links back to its `PolicyEntity_Ext` instances. A policy can have many policy extensions, but each policy extension belongs to only one policy.

To ensure that `PolicyEntity_Ext` is part of the policy graph, modify the policy refresh plugin to include it in the set of entity types that comprise the policy graph. Thus, ClaimCenter knows that your new `PolicyEntity_Ext` entity is part of the policy graph.

Now, suppose your new `PolicyEntity_Ext` has a foreign key to `Exposure`. The `Exposure` entity is a built-in type that belongs to the claim graph. In turn, you extend `Exposure` with an array property that contains `PolicyEntity_Ext` instances. An exposure can have many policy extensions, but each policy extension belongs to only one exposure.

Example of extending a policy-specific entity metadata definitions

You define your new entity `PolicyEntity_Ext` in the following metadata definition file.

`ClaimCenter/config/extensions/PolicyEntity_Ext.eti`

The contents of the file might look like the following example.

```
<?xml version="1.0"?>
<entity
  xmlns="http://guidewire.com/datamodel"
  entity="PolicyEntity_Ext"
  table="policyentity"
  exportable = "true"
  type="retireable">
  <implementsEntity name="Extractable"/>
  <foreignkey name="Policy" fkentity="Policy"/>
  <foreignkey name="Exposure" fkentity="Exposure"/>
  <column
    name="Name"
    type="shorttext"
    desc="Policy Entity Name"/>
</entity>
```

In `ClaimCenter/config/extensions/Policy.etcx`, extend the `Policy` entity with an array of policy extensions.

```
<array name="PolicyEntity_Exts" arrayentity="PolicyEntity_Ext"/>
```

In `ClaimCenter/config/extensions/Exposure.etcx`, extend the `Exposure` entity with an array of policy extensions.

```
<array name="PolicyEntity_Exts" arrayentity="PolicyEntity_Ext"/>
```

The changes described in this example are common types of data model changes. Because your new entity belongs to the policy graph, the policy refresh system requires you to perform additional configuration tasks beyond those associated with standard data model extensions.

- Modify the policy refresh plugin to include `PolicyEntity_Ext` in the set of entity types that comprise the policy graph.

- Modify your policy administration system to correspond to the changes you make in the ClaimCenter policy graph.
- Perform configuration tasks in other parts of the policy refresh system to complete your addition of new policy-related entities to the ClaimCenter policy graph.
 - Adding new policy-related entities to the policy refresh display tree
 - Creating custom matcher classes
 - Creating customer relinker classes

How policy refresh updates objects in the claim and policy graphs

When policy refresh happens, ClaimCenter replaces any `PolicyEntity_Ext` entity instances, which belong to the policy graph. For foreign key links from `PolicyEntity_Ext` instances to exposures, the policy refresh system relinks them automatically. The directionality of the foreign key runs from a policy object to a claim object.

Foreign key links from updated `PolicyEntity_Ext` instances to other updated policy-specific instances in the refreshed policy graph do not require relinking. The policy refresh system retrieves valid links between policy-specific instances from the policy administration system whenever policy refresh refreshes a policy snapshot in ClaimCenter.

Modifying your policy system for changed entities in the policy graph

Whenever you add a new entity type to the policy graph in ClaimCenter, policy refresh requires you to perform additional configuration tasks beyond those associated with standard data model extensions. Tasks depend on whether your policy administration system is PolicyCenter or a third-party system.

Tasks in PolicyCenter for new policy-specific entities

Whenever you add a new entity type to the policy graph in Claim Center, you must create a corresponding new entity type in PolicyCenter. This topic continues the example of adding the `PolicyEntity_Ext` entity type to the policy graph in ClaimCenter.

Add new policy-specific entity types to the PolicyCenter data model

In PolicyCenter Studio, define a new `PolicyEntity_Ext` entity in the following metadata definition file.

`PolicyCenter/config/extensions/PolicyEntity_Ext.eti`

The contents of the PolicyCenter metadata definition file differ slightly from the ClaimCenter version. In PolicyCenter, the `PolicyPeriod` entity type corresponds to the ClaimCenter `Policy` entity type. So, the foreign key elements in the two versions differ.

```
<?xml version="1.0"?>
<entity
  xmlns="http://guidewire.com/datamodel"
  entity="PolicyEntity_Ext"
  table="policyentity"
  exportable = "true"
  type="retireable">
  <implementsEntity name="Extractable"/>
  <foreignkey name="PolicyPeriod" fkentity="PolicyPeriod"/>
  <column
    name="Name"
    type="shorttext"
    desc="Policy Entity Name"/>
</entity>
```

In the file `PolicyCenter/config/extensions/PolicyPeriod.etx`, extend the `PolicyPeriod` entity with array of policy extensions.

```
<array arrayentity="PolicyEntity_Ext" cascadeDelete="true" name="PolicyEntities_Ext"/>
```

Add a Gosu class for each new policy-specific entity type

In PolicyCenter Studio, create or modify Gosu classes to match your changes to the data model. Prefix the class names with CC. Continuing our example from above, create a Gosu class called `CCPolicyEntity_Ext`.

```
package gw.webservice.pc.pcVERSION.ccintegration.ccentities
class CCPolicyEntity_Ext {
    var _name : String as Name
    construct() {
    }
}
```

The purpose of the Gosu class `CCPolicyEntity_Ext` is to mirror the `PolicyEntity_Ext` entity in ClaimCenter. Follow this pattern for all custom policy graph entities.

Note the following qualities of these special CC classes.

- They do not need public property getters and setters.
- They have an empty constructor.

Modify the policy generator and ClaimCenter policy classes

In PolicyCenter Studio, modify `CCPolicyGenerator`, specifically the `generatePolicy` method. Change the method to instantiate the new Gosu classes and copy data from the persistent entities as needed. Continuing the example of the entity `PolicyEntity_Ext` and the Gosu class `CCPolicyEntity_Ext`, the modified `generatePolicy` method looks like the following sample Gosu code.

```
uses gw.webservice.pc.pcVERSION.ccintegration.ccentities.CCPolicy
/*
 * Generate a ClaimCenter Policy object and related objects from a PolicyCenter PolicyPeriod.
 */
class CCPolicyGenerator {
    ...
    /*
     * Create the ClaimCenter Policy and all related objects for the given PC PolicyPeriod.
     */
    public function generatePolicy( pcPolicyPeriod : PolicyPeriod ) : CCPolicy {
        ...
        // Generate ClaimCenter versions of custom PolicyCenter entities.
        for (pcEntity in pcPolicyPeriod.PolicyEntities_Ext) {
            ...
            // Instantiate a ClaimCenter instance.
            var newCCPolicyEntity_Ext = new CCPolicyEntity_Ext()

            // Copy properties from the PolicyCenter instance to the ClaimCenter instance.
            newCCPolicyEntity.Name = pcEntity.Name

            // Add the populated ClaimCenter instance to the ClaimCenter policy instance.
            _policy.addToPolicyEntities_Ext(newCCPolicyEntity_Ext)
        }
        return _policy
    }
}
```

Add the adder function in the Gosu class `CCPolicy.gs`.

```
class CCPolicy {
    var _policyEntities = new ArrayList<CCPolicyEntity_Ext>()
    ...
    function addToPolicyEntities_Ext(policyEntity : CCPolicyEntity_Ext) : void {
        _policyEntities.add(policyEntity)
    }
}
```

Start the PolicyCenter application to verify your data model extensions and Gosu modifications.

Refresh the WSDL in ClaimCenter

In Guidewire Studio for ClaimCenter, refresh the WSDL from PolicyCenter. PolicyCenter recursively reflects on the CCPolicy Gosu class and its related types, including the new ClaimCenter-related Gosu classes you added. There now will be new XSD types in the WSDL files that allow ClaimCenter to retrieve policies from PolicyCenter. ClaimCenter now knows from the WSDL about the new Gosu classes that you added to PolicyCenter, such as `CCPolicyEntity_Ext`.

By default, ClaimCenter automatically copies data from a ClaimCenter Gosu object (in our example, `CCPolicyEntity_Ext`) to the corresponding entity (in our example, `PolicyEntity_Ext`). The copying occurs only if the following conditions are met.

- The type names match – Other than the prefix CC on the Gosu type within ClaimCenter
- The type properties match – Specifically, matching names and types of fields

The rules listed above are only the default rules. You can modify the rules to do a custom mapping by modifying the file `pc-to-cc-data-mapping.xml`. For example, you could not require the CC prefix for some entity types, or you could do some other custom matching.

Tasks in third-party policy administration systems for new policy-specific entities

If you use a policy administration system other than PolicyCenter, use the `BasicPolicyRefreshConfiguration` configuration class in ClaimCenter to include new policy-specific entity types in the policy refresh system.

In the class `BasicPolicyRefreshConfiguration`, modify the `getPolicyOnly` method. In our continuing example, the changes look like the following.

```
class BasicPolicyRefreshConfiguration extends ExtendablePolicyRefreshConfiguration {  
    ...  
  
    /*  
     * This method extracts all Policy-only entities from the existing Policy (which is linked  
     * to a Claim and other non-Policy entities).  
     */  
    override function getPolicyOnly(existingPolicy : Policy) : Set<KeyableBean> {  
        // Use helper method getEntireArray to include retired entities, if entity is retireable  
        getEntireArray(existingPolicy, "PolicyEntities_Ext", PolicyEntity_Ext).each(  
            \ g -->includePolicyEntity_Ext(policyOnly, g))  
  
        return policyOnly  
    }  
}
```

Configuring policy refresh for new policy-specific entities

Whenever you add a new entity type to the policy graph, you must perform additional configuration tasks related to policy refresh, regardless what policy administration system you use. If you performed the preceding tasks properly, custom entities in the policy administration system become part of the new policy graph in ClaimCenter and are replaced when appropriate.

You must perform the following additional configuration tasks in other parts of the policy refresh system to complete your addition of new policy-related entities to the ClaimCenter policy graph.

Customizing the policy refresh display tree for new policy-specific entities

Whenever you add custom entities to the policy graph in ClaimCenter, you must add the custom entities to the policy refresh display tree in the **Policy Refresh Wizard**. The wizard compares the differences between the current snapshot of a policy in ClaimCenter and a refresh of the policy snapshot from the policy administration system.

Optionally create custom `DiffDisplay` objects to customize the appearance of node entries for custom entities in the policy refresh comparison screen. Create a custom `DiffDisplay` and register it with `PolicyRefreshConfigurationBase`. Using our continuing example, create the following Gosu class and note the use of Gosu generics in the first line.

```
class PolicyEntity_ExtDiffDisplay extends EntityDiffDisplayBase<PolicyEntity_Ext> {
```

```

construct(theDiff : EntityDiff<ClassCode>, theType : DiffDisplay.Type) {
    super(theDiff, theType)
}

override function getMessages(ctx : PolicyRefreshMessageContext) : UIMessageList {
    var result = new UIMessageList()
    if (Type==REMOVED) {
        result.add(UIMessage.error("hello I am an error"))
    }
    return result;
}
}

```

Next, add your new `DiffDisplay` class to the `DiffDisplayTypes` property map in `PolicyRefreshConfigurationBase`, as the following sample Gosu code shows.

```

override property get DiffDisplayTypes() : Map<IEntityType, Class<DiffDisplay>> {
    return {
        ...
        PolicyEntity_Ext -> PolicyEntity_ExtDiffDisplay,
        ...
    }
}

```

Next, in `PolicyRefreshConfigurationBase`, modify the getter for the `DisplayTree` property so it includes a reference to new custom entities in their appropriate places in the policy graph. For instance, if the customization introduces a new `PolicyEntity_Ext[]` array property on `Policy` called `AllPolicyEntities` that refers to `PolicyEntity_Ext` entities, then you would add the following association.

```
tree.addOneToMany(policyRoot, "AllPolicyEntities", PolicyEntity_Ext)
```

Note that the association must satisfy the following conditions.

- The cardinality of the property must match – If it is an array property, use the method `addOneToMany`. If it is a one-to-one property, use the method `addOneToOne`.
- The property name must match – If the data model property is called `AllPolicyEntities`, then you must pass the string `"AllPolicyEntities"` as the second parameter to `addOneToMany` (or to `addOneToOne`).
- The property type must match or be a supertype – If the property is `PolicyEntity_Ext`, then the third parameter must either be `PolicyEntity_Ext`, or a supertype of `PolicyEntity_Ext`.

[Customizing policy refresh matchers for new policy-specific entities](#)

While technically not necessary, it is a practical requirement that you write your custom matchers in Gosu for custom policy graph entities. Next, register your matchers in the policy refresh configuration file. Without a custom matcher, the default behavior relies on object identity to determine if a given current policy-related entity instance matches a given refreshed policy-related entity instance.

Object identity is almost always an inappropriate way to match policy-related entity instances. Instead, create a Gosu class for your own matcher. The class you extend to create your class depends on your matching requirements for your new policy-related entity.

- If your matcher matches instances of your new entity only on its own properties and not the properties of related entity types, extend the class `gw.api.bean.compare.MatcherBase`.
- If your matcher matches instances of your new entity on properties of related entity types, extend the class `gw.api.bean.compare.InitializableMatcherBase`.

The following example Gosu matcher class extends the `InitializableMatcherBase` class.

```

package gw.plugin.policy.refresh.matcher
uses gw.api.bean.compare.InitializableMatcherBase

/*
 * Entity matcher for a PolicyEntity_Ext.
 */
class PolicyEntity_ExtMatcher extends InitializableMatcherBase<PolicyEntity_Ext> {

    override function doEntitiesMatch(p0 : PolicyEntity_Ext, p1 : PolicyEntity_Ext) : boolean {

```

```
        return p0.Name.equals(p1.Name)
    }
```

Next, register your new matcher class with policy refresh by adding it to the `MatcherTypes` property map in `PolicyRefreshConfigurationBase`, as in this example.

```
override property get MatcherTypes() : Map<IEntityType, Class<EntityMatcher<KeyableBean>>> {
    return {
        ...
        PolicyEntity_Ext -> PolicyEntity_ExtMatcher,
        ...
    }
}
```

Customizing policy refresh relinkers for new policy-specific entities

Whenever you add new policy-specific entities to the policy graph in ClaimCenter, create custom Relinker classes for them. The default relinking behavior suffices for most cases without requiring a custom Relinker. If a custom entity does require special relinking behavior, create a custom Relinker and register it with `PolicyRefreshConfigurationBase`.

```
package gw.plugin.policy.refresh.relink.handler

uses gw.api.policy.refresh.relink.PerLinkHandler
uses gw.api.policy.refresh.relink.IgnoreIfNotMatchedLinkHandler

class PolicyEntity_ExtRelinkHandler extends PerLinkHandler<PolicyEntity_Ext> {

    construct() {
        register(PolicyEntity_Ext, "PolicyEntity_Ext",
            new IgnoreIfNotMatchedLinkHandler<PolicyEntity_Ext>())
    }
}
```

The `CustomRelinkerTypes` property getter returns a map. In the file `PolicyRefreshConfigurationBase`, add a row like the following code block.

```
override property get CustomRelinkerTypes() : Map<IEntityType, Class<RelinkHandler>> {
    return {
        ...
        PolicyEntity_Ext -> PolicyEntity_ExtRelinkHandler,
        ...
    }
}
```


Contact integrations

ClaimCenter integrates with external systems to manage information about contacts. ClaimCenter uses integrations to send information to or receive information from the external systems. Examples of such information are requests for contact information that ClaimCenter sends to a contact management system and information about changes to contact details that a contact management system sends to ClaimCenter.

ClaimCenter uses both web services and plugins for data transfer.

Contact integration

ClaimCenter supports integration with an external contact management system other than Guidewire ContactManager.

The base configuration of ClaimCenter includes support for integrating with Guidewire ContactManager.

See also

- *Contact Management Guide*

Integrating with a contact management system

ClaimCenter integrates with an external contact management system through the `ContactSystemPlugin` plugin interface. This plugin interface integrates with an external system that supports create, retrieve, update, search, and finding duplicates. To integrate with an external contact management system, write your own implementation of the `ContactSystemPlugin` plugin and register it in Guidewire Studio™.

If you integrate with ContactManager, do not write your own implementation. For integration with ContactManager, ClaimCenter provides an implementation of the `ContactSystemPlugin` plugin interface, `gw.plugin.contact.ab1000.ABContactSystemPlugin`.

Your implementation of the contact system plugin defines the contract between ClaimCenter and the code that initiates requests to an external contact management system. ClaimCenter relies on this plugin to perform the following operations.

- Retrieve a contact from the external contact management system.
- Search for contacts in the external contact management system.
- Add a contact to an external contact management system.
- Update a contact in an external contact management system.

ClaimCenter uniquely identifies contacts by using unique IDs in the external system, known as the Address Book Unique ID (AddressBookUID). This unique ID is analogous to a public ID, but is not necessarily the same as the public ID, which is the separate property `PublicID`.

When ClaimCenter needs to retrieve a contact from the contact management system, it calls one of the `retrieveContact` methods of this plugin. The method called depends on whether ClaimCenter needs to retrieve any related contacts along with the contact being retrieved. In both cases, ClaimCenter passes the address book unique ID of the contact. If related contacts are required, a list of relationships to retrieve is passed in an alternate method signature. Your plugin implementation needs to get the contact from the external system by using whatever network protocol is appropriate. Your plugin implementation must then populate either a `Contact` object or a graph that possibly includes related contacts.

For this plugin, the contact record object that you must populate is a `Contact` entity instance. An additional method, `retrieveRelatedContacts`, retrieves the related contacts as specified in the collection of relationships passed in. This method must return those related contacts merged into the contact graph that was passed in.

If you need the ClaimCenter implementation to retrieve additional fields from the external contact management system, the best practice is to extend the data model for the `Contact` entity.

If you need to view or edit these additional properties in ClaimCenter, you must modify the contact-related PCF files to extract and display the new field.

Inbound contact integrations

For inbound integration, ClaimCenter publishes a WS-I web service called `ContactAPI`. This web service enables an external contact management system to notify ClaimCenter of updates, deletes, and merges of contacts in that external system. When ClaimCenter receives a notification from the external contact management system, ClaimCenter can update its local copies of those contacts to match.

See also

- “Contact web service APIs” on page 540

Asynchronous messaging with the contact management system

When a user in ClaimCenter creates or updates a local `Contact` instance, ClaimCenter runs the Event Fired rule set. These rules determine whether to create a message that creates or updates the `Contact` in the contact management system. In the base configuration, the process of sending the message to the contact system involves several messaging objects.

- A built-in `ContactSystemPlugin` plugin implementation.
- A built-in messaging destination that is responsible for sending messages to the contact system. The messaging destination uses two plugins.
 - A message transport (`MessageTransport`) plugin implementation called `ContactMessageTransport`. ClaimCenter calls the message transport plugin implementation to send a message.
 - A message request (`MessageRequest`) plugin implementation called `ContactMessageRequest`. ClaimCenter calls the message request plugin to update the message payload immediately before sending the message if necessary.

In the default configuration, ClaimCenter calls the following two methods of `ContactSystemPlugin`.

- `createAsyncUpdate` – At message creation time, Event Fired rules call this method to create a message.
- `sendAsyncUpdate` – At message send time, the message transport calls this method to send the message.

Detailed contact messaging flow

1. At message creation time, the Event Fired rules call the `ContactSystemPlugin` method called `createAsyncUpdate` to actually create the message. The Event Fired rules pass a `MessageContext` object to the `ContactSystemPlugin` method `createAsyncUpdate`. The `createAsyncUpdate` method uses the `MessageContext` to determine whether to create a message for the change to the contact, and, if so, what the payload is for that message.
2. At message send time on a server with the `messaging` server role, ClaimCenter sends the message to the messaging destination. There are two phases of this process.
 - a. ClaimCenter calls the message request plugin to handle late-bound `AddressBookUID` values. For example, suppose the `AddressBookUID` for a contact is unknown at message creation time but is known when at message send time. ClaimCenter calls the `beforeSend` method of the `ContactMessageRequest` plugin to update the payload before the message is sent.
 - b. ClaimCenter calls the message transport plugin to send the message. The message transport implementation finds the `ContactSystemPlugin` class and calls its `sendAsyncUpdate` method to actually send the message.

An additional method argument includes the modified late-bound payload that the message request plugin returned.

ClaimCenter synchronizing and mapping

The `ContactSystemPlugin` provides methods enabling ClaimCenter to access the properties that have been defined as synchronizable and persistent in the ClaimCenter mapping files.

- `getSyncablePropertiesForType` — Returns the properties that have been defined as affecting the sync status in ClaimCenter.
- `getPersistPropertiesForType` — Returns the properties that have been defined as persistent in ClaimCenter.
- `getAllMappedPropertiesForType` — Returns the properties mapped between ClaimCenter and ContactManager.
- `getSyncableRelationshipsForContactType` — Returns the set of relationships that have been defined as synchronizable for a given Contact type.

Retrieve a contact

You use a web service and the contact system plugin to retrieve a contact from an external system.

About this task

To support contact retrieval, do the following steps in the web service and the `retrieveContact` method of the contact system plugin.

Procedure

1. Write your web service code that connects to the external contact management system.
2. In the web service, write a method to return a data transfer object that contains enough information to populate a contact.

The return type must contain equivalents of the following `Contact` properties.

```
PrimaryPhone  
TaxID  
WorkPhone  
EmailAddress2  
FaxPhone  
HomePhone  
CellPhone  
DateOfBirth  
Version
```

A contact of type `Person` must also have the properties `FirstName` and `LastName`.

A contact of type `Company` must also have the property `Name`, containing the company name.

Consider a web service in Studio with the name `MyContactSystem` and with a `MyContact` XML type from a WSDL or XSD file. For this example, the contact retrieval web service returns a `MyContact` data transfer object.

3. In your plugin code for the `retrieveContact` method, wait synchronously for a response.
4. To return the data to ClaimCenter, create a new instance of a contact in the current transaction's bundle.
5. Populate the new contact entity instance with information from your external contact.
6. Return that object as the result from your `retrieveContact` method.

Retrieve a persistable contact

To retrieve a persistable Contact object from the contact system, call the static method `importContactFromContactSystem` in the `ContactSystemUtil` class. Note: A Contact object retrieved using the `retrieveContact` method of the `ContactSystemPlugin` is not persistable.

```
static importContactFromContactSystem(abUIDOfContact : String, syncRelatedContacts : boolean) : Contact
```

The method accepts the following arguments.

- `abUIDOfContact` – AddressBookUID of the contact to retrieve.
- `syncRelatedContacts` – If true, include all related contacts with the retrieved contact.

The method returns a persistable local copy of the desired contact. If the `syncRelatedContacts` argument is true, any contacts related to the retrieved contact are included in the returned Contact object's `RelatedContact` property.

In the event of an error, the method can throw the following exceptions.

- `ContactSystemNotFoundException` – The contact specified by the `abUIDOfContact` argument was not found in the contact system.
- `ContactSystemNotAvailableException` – The contact system cannot be contacted or has not been configured.
- `ContactSystemPluginException` – To perform its task, the `importContactFromContactSystem` method calls the `ContactSystemPlugin retrieveContact` method. If the `ContactSystemPlugin` is not defined, this exception is thrown.

Contact searching

To support contact searching, your plugin must respond to a search request from ClaimCenter. ClaimCenter calls the plugin method `searchContacts` to perform the search. The details of the search are defined in a contact search criteria object, `ContactSearchCriteria`, which is a method argument. This object defines the fields that the user searched on.

The important properties in this object are listed below.

- `ContactIntrinsicType` – The type of contact. To determine if the contact search is for a person or a company, use code similar to the following example.

```
var isPerson = Person.Type.isAssignableFrom(searchCriteria.ContactIntrinsicType)
```

If the result is true, the contact is a person rather than a company.

- `FirstName`
- `Keyword` – The general name for a company name (for companies) or a last name (for people)
- `TaxID`
- `OrganizationName`
- `Address.AddressLine1`
- `Address.City`
- `Address.State.Code`
- `Address.PostalCode`
- `Address.Country.Code`
- `Address.County`

Refer to the *Data Dictionary* for the complete set of fields on the search criteria object that you could use to perform the search. However, the built-in implementation of the user interface might not necessarily support populating those fields with non-null values. You can modify the user interface code to add any existing fields or extend the data model of the search criteria object to add new properties.

The second parameter to this method is the `ContactSearchFilter`, which defines the following metadata about the search.

- The start row of the results to be returned
- The maximum number of results, if you are just querying for the number of results, as opposed to the actual results
- The sort columns
- Any subtypes to exclude from the search due to user permissions

For the return results, populate a `ContactResult` object, which includes the number of results from the query and, if required, the actual results of the search as `Contact` entities. It is not expected that these `Contact` entities will contain all the contact information from the external contact management system. These entities are expected to contain just enough information to display in a list view to enable the user to select a result.

For example, the default configuration of the plugin for ContactManager includes the following properties on the `Contact` entities returned as search results.

- `AddressBookUID` – The unique ID for the contact as `String`
- `FirstName` – First name as `String`
- `LastName` – Last name as `String`
- `Name` – Company or place name as `String`
- `DisplayAddress` – Display version of the address, as a `String`
- `PrimaryAddress` – An address entity instance
- `CellPhone` – Mobile phone number as `String`
- `HomePhone` – Home phone number as `String`
- `WorkPhone` – Work phone number as `String`
- `FaxPhone` – Fax phone number as `String`
- `EmailAddress1` – Email address as `String`
- `EmailAddress2` – Email address as `String`

For the full list of properties, see the `ABContactAPISearchResult` Gosu class in ContactManager. This class is in the package `gw.webservice.ab.ab1000.abcontactapi`.

See also

- [Contact Management Guide](#)

Finding duplicate contacts

ClaimCenter calls the `findDuplicates` method in the plugin to find duplicate contacts for a specified contact.

The method takes two arguments.

- A `Contact` entity instance for which you are checking for duplicates
- A `ContactSearchFilter` that can specify subtypes to exclude from the results if contact subtype permissions are being used

The `findDuplicates` method must return an instance of `DuplicateSearchResult`, a class that contains a collection of the potential duplicates as `ContactMatch` objects and the number of results.

The `ContactMatch` object contains the `Contact` that was found to be a duplicate and a boolean property, `ExactMatch`, which indicates if the contact is an exact or a potential match. As with searching, the `Contact` returned in the `ContactMatch` object does not have to contain all the contact information from the external contact management system. The object contains just enough contact information to enable the ClaimCenter user to determine if the duplicate is valid.

The list of properties returned by ContactManager in its base configuration is in the `ABCContactAPIFindDuplicatesResult` class in the package `gw.webservice.ab.ab900.abcontactapi`.

See also

- *Contact Management Guide*

Find duplicate contacts

You use an instance of the concrete class `DuplicateSearchResult` to investigate potentially duplicate contacts that an external system stores.

Procedure

1. Query the external system for a list of matching or potentially matching contacts.
2. Optionally, if the results you receive are only summaries, if it is necessary for detecting duplicates, retrieve all the data for each contact from the external system.
3. Review the duplicates and determine which contacts are duplicates according to your plugin implementation.
4. Create a list of `ContactMatch` items, each of which contains a `Contact` and a boolean property that indicates if the contact is an exact match. Each of these `Contact` objects is a summary that contains many, but perhaps not all, `Contact` properties.
5. Create an instance of the concrete class `DuplicateSearchResult` with the collection of `ContactMatch` items. Pass the list of duplicates to the constructor.
6. Return that instance of `DuplicateSearchResult` from this method.

Adding contacts to the external system

To support ClaimCenter sending new contacts to the external contact management system, implement the `createContact` methods in your contact system plugin. This method must add the contact to the external system. Send as many fields on `Contact` as make sense for your external system. If you added data model extensions to `Contact`, you must decide which ones apply to the external contact management system data model. There may be side effects on the local contact in ClaimCenter, typically to store external identifiers.

Methods for adding a contact to an external system

The `createContact` method has two signatures.

```
createContact(Contact contact)
createContact(Contact contact, String transactionId)
```

The simpler method signature takes only a `Contact` entity.

The transaction ID parameter uniquely identifies the request. Your external system can use this transaction ID to track duplicate requests from ClaimCenter.

Capturing the contact ID from the external system

The `createContact` method returns a `ContactCreateResult` object to ClaimCenter. The object contains the `Contact` and a flag to indicate if the contact is in a pending create status in the external contact management system. Your implementation of the method must capture the ID for the new contact and for other entities in the contact graph from the external system. Use the captured value to set the address book UID on the local ClaimCenter contact and other entities in the contact graph. You must set the address book UID before your `createContact` method returns control to the caller. Typically, the external system is the system of record for issuing address book UIDs.

Updating contacts in the external system

If a ClaimCenter user updates a contact's information, ClaimCenter sends the update to the contact management system by calling the `updateContact` method of the contact system plugin. The method accepts a parameter of a

Contact entity. The second version of the method adds a transaction ID that can be used by the external contact management system to track if an update has already been applied.

Configuring contact links

You can change the behavior when linking contacts to an external contact management system using the `ContactSystemLinkPlugin` plugin interface. ClaimCenter includes a default implementation called `ABCContactSystemLinkPlugin`. The default implementation uses the existing `ContactSystemPlugin` implementation to actually initiate requests to the external contact management system.

There are two plugin methods to implement: `linkContact` and `link`.

The `linkContact` method

ClaimCenter calls the `linkContact` plugin method when ClaimCenter needs to link a contact to an external contact management system. The method takes the following arguments.

- A ClaimCenter `Contact` entity instance
- A boolean flag that indicates whether to commit the change immediately. If `true`, explicitly commit the entity change in the `Contact` object's current bundle. If `false`, do not commit the change because the caller code is responsible for the bundle commit.

The method returns a `LinkResult` object that encapsulates the link status and relevant information. Status types include the following, which are static fields on the `LinkResult` class.

The following table lists the status options, and the name of the static method you can use to create a `LinkResult` object of that type.

Status	Description	Static method name	Static method arguments, which also become data inside the <code>LinkResult</code>
CREATED	The contact was created in the external system	<code>created</code>	• the address book UID
PENDING_CREATE	The contact is in pending-create status.	<code>pendingCreate</code>	• the address book UID
EXACT	The contact is an exact match with a <code>exact</code> contact in the external system.		• the address book UID
POTENTIAL_MATCH	The contact is a potential match with <code>potential</code> a contact in the external system.		• potential matches, as a <code>DuplicateContactMatch</code>
ERROR	Some error	<code>error</code>	n/a

The default implementation tries several tactics, choosing the first attempt that succeeds.

- Search for exact matches
- If none found, search for potential matches
- If none found, create a new contact in the external system

In the base implementation, whether a contact is created or marked as pending-create will be determined by the utility class `ContactSystemApprovalUtil`. The contact system plugin built-in plugin implementation calls this utility class when attempting to create a contact in `ContactManager`.

The `linkContact` method must first determine whether to create the contact in remote contact management system, use an definitively matched contact, or pick from a list of potentially matched contacts. In the default implementation, the `linkContact` method calls its own `link` method to perform the actual linking of the contact to the remote contact management system.

The link method

ClaimCenter calls the `link` plugin method when it tries to link two contacts together.

In the default implementation, it sets the `AddressBookUID` on the ClaimCenter `Contact` to the value in the external contact management system, `ContactManager`.

The method takes the following arguments.

- a contact, as a ClaimCenter `Contact` entity instance
- The address book UID of the contact in the external contact management system

The method has no return value.

Unlinking contacts

There is no method in the plugin interface specifically for unlinking a contact from the external system, but you may need to unlink the contact if the link is broken.

In the base implementation, if the `link` method's address book UID parameter is `null`, the `link` method unlinks the contact.

If you need to unlink a contact, you must remove all the address book UID values throughout the contact graph with the exception of related contacts. If you need to implement similar logic, review the base implementation code for guidance.

Contact web service APIs

The web service `ContactAPI` provides external systems a way to interact with contacts in ClaimCenter. Refer to the implementation class in Guidewire Studio™ for details of all the methods in this API. Each of the methods in this API that changes contact data in ClaimCenter requires a transaction ID, which you must set in the SOAP request header for the call.

Deleting a contact

Before a contact management system deletes a contact that is in ClaimCenter, it must first query to see if the contact is in use. The `isContactDeletable` method checks to see if a contact with the address book UID passed in is currently in use in ClaimCenter. ClaimCenter uses a batch process to retire contacts that are not in use. Therefore, this method can return `false` and then sometime later return `true`, after the batch process has run.

If the call to `isContactDeletable` determines that the contact can be deleted, the external contact management system calls the `removeContact` method. This method indicates to ClaimCenter that the contact has been deleted from the external contact management system. The `removeContact` method then attempts to retire all the contacts with that address book UID in ClaimCenter.

Updating a contact

You can update a contact from an external system by using the `updateContact` method. The method has no return value.

Because ClaimCenter can have many local instances of a `Contact`, `updateContact` uses the `ContactAutoSync` process to cause ClaimCenter to copy over the latest version from the contact management system. Depending on the configuration of the `ContactAutoSync` process, this update might happen immediately or might happen when the batch process is run.

Merging contacts

You can merge two contacts from an external system. You need to know the IDs of both contacts, and you must decide which one will survive after the merge.

To merge contacts, use the `mergeContacts` method. This method has no return value.

Pass the following parameters in the listed order.

1. The ID of the contact to keep. This parameter is known as the “kept” contact.
2. The ID of the contact to delete. This parameter is known as the “deleted” contact.

Merging contacts causes the `AddressBookUID` on the contacts in ClaimCenter that match the deleted contact to be changed to that of the kept contact. They are then copied by using the `ContactAutoSync` process. As with `updateContact`, the timing of this update depends on the configuration of the `ContactAutoSync` process.

Handling rejection and approval of pending changes

An external contact management system might, like ContactManager, support pending changes. Pending changes are changes to contacts that are applied in the core application, but require approval in the contact management system before being applied there.

In the base configuration, BillingCenter and PolicyCenter do not use the pending changes feature. All changes to contacts in either of these core applications are applied in the contact management system. BillingCenter and PolicyCenter implement the pending change methods, as required by `ABCClientAPI`, and if a contact management system calls any of these methods, the application throws an exception.

The ClaimCenter implementation of `ContactAPI` has methods that the external contact management system can call to indicate if pending create or pending update operations have been approved or rejected. In each case, the methods pass in a parameter that contains the context of the original change, usually the user, claim, and contact information. This information enables ClaimCenter to notify the user of the results of the operation. If the change is rejected, ClaimCenter creates an activity for the user giving the details of the change that was rejected. If the rejection was for a pending update, ClaimCenter also copies the contact data from the contact management system and attaches it as a note to the activity.

These pending change methods are listed below.

- `pendingCreateRejected` – ClaimCenter creates a pending create rejected activity for the user that created the contact.
- `pendingCreateApproved` – No action is taken. ClaimCenter already has the `AddressBookUID` for the contact.
- `pendingUpdateRejected` – ClaimCenter creates a pending update rejected activity for the user that changed the contact.
- `pendingUpdateApproved` – ClaimCenter updates the contact graph with any new `AddressBookUID` values that were created for any new entities that were created in the update. Additionally, if there was context information sent with the update approval, ClaimCenter synchronizes all contacts that have this `AddressBookUID` with the contact management system.

part 8

Importing claims data

Importing claim data

ClaimCenter supports bulk data import by using database staging tables. A typical use for staging tables is migrating claims from a legacy system to ClaimCenter. ClaimCenter also supports the use of staging tables to import zone data. ClaimCenter uses zone data for functionality related to particular locations, including address auto-fill.

The ClaimCenter database contains many related tables that describe claims and all supporting information. Transferring large quantities of data from an external source to a ClaimCenter production database is a complex procedure. Staging tables provide an intermediate data location between the external source and the operational tables in the database. You write a conversion tool to insert rows based on the external data into the staging tables. You use Guidewire tools to check the integrity and consistency of the data in the staging tables. Correcting errors in the staging table data reduces operational down time caused by rolling back failed data imports. You complete the import process by using Guidewire tools to load the data into the operational tables. This final step uses bulk SQL Insert and Select statements that operate on entire tables to ensure high performance.

Importing zone data

You can also import zone data from a comma-separated values (CSV) file to a staging table. You set up the CSV file to contain the data columns you need. Guidewire provides tools to load the CSV data into the staging table.

ClaimCenter uses zone data for the following features.

- Assignment by location
- Address auto-fill
- Setting regional holidays
- Defining catastrophes

Zone data information changes frequently. As population changes in an area, location boundaries change and new locations appear. You need to load updated zone data into ClaimCenter regularly.

Importing FNOL data

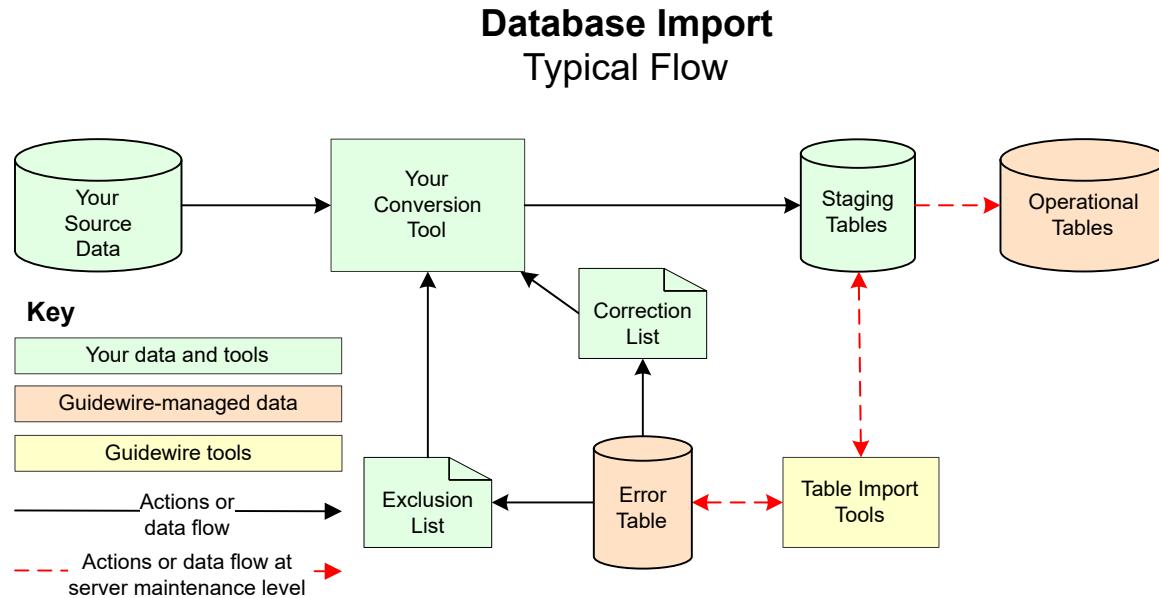
You can also import First Notice of Loss (FNOL), or initial claim, reports by using the FNOL mapper. Typically, the FNOL mapper imports FNOL report data from XML documents in the standard XML format known as ACORD XML for Property & Casualty. However, you can configure the FNOL mapper to import report data in other custom XML formats.

Importing from database staging tables

You use database staging tables and the database import tools to migrate claims from your legacy system to ClaimCenter. As a prerequisite to migrating legacy claims, you must develop a conversion tool that selects claims from your legacy system and transforms their data to ClaimCenter format. Then, your tool must insert the converted data into the staging tables.

High-level steps in a typical database staging table import

The following diagram shows the major steps in a typical database staging table import.



The following procedure lists the high-level steps involved in a typical database import procedure to migrate claim data from your legacy system to ClaimCenter.

Create and run your conversion tool

This tool selects and converts external claim data to ClaimCenter format and inserts the converted data into staging tables.

Check the data in the staging tables

Set the server to the maintenance run level. Run integrity checks on staging table data. Set the server to the multiuser run level.

Review and correct data errors

View load errors on the [Load Errors](#) page. Fix the errors and then repeat the previous tasks until no errors remain.

Load data into operational tables

Set the server to the maintenance run level. In a cluster installation, shut down all but one server. Load the data from the staging tables into operational tables by using ClaimCenter web services or command prompt table import tools. Set the server to the multiuser run level.

Importing contact data

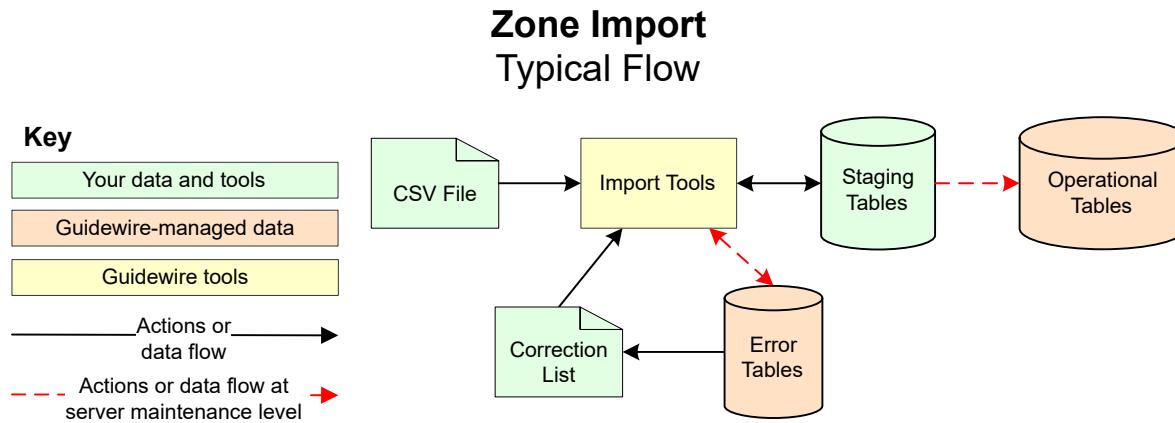
If ClaimCenter uses ContactManager to store contact data, you can load contact data into the ContactManager database in the same way as you load data into the ClaimCenter database. The staging tables that contain shared contact data in the ContactManager database have names that start with ab_AB.

Importing zone data

You can import zone data by using database staging tables and the database import command. The sources for zone data are CSV files for each country or region that you need to support. ClaimCenter provides both a command prompt tool and a web services API to load zone data from CSV files into the staging table for zone data. Configuration files for this tool specify the zone fields to import.

Guidewire recommends that you load a complete set of data for a country rather than loading incremental changes to zone data. Incremental changes cause a non-linear growth in the time to import zone data and in the size of the zone link table. To reload data for a country for which zone data already exists, you must first clear the zone data for that country from the operational zone tables.

The following diagram shows the major steps in a typical zone staging table import.



High-level steps to import zone data

The following procedure lists the high-level steps involved in a typical database import procedure to load zone data from CSV files into ClaimCenter.

- Get or create zone data files – These files use comma separated value (CSV) format. Each line in a file contains the columns that the country or region requires, such as the postal code, state, city, and county. A configuration file for the country or region specifies the columns and their order in the line.
- Run the zone import command to add zone data to the zone staging table – Run a command prompt tool or use the web services API to add your zone data to the staging table.

- Check the data in the zone staging tables – Set the server to the maintenance run level. Run integrity checks on staging table data. Set the server to the multiuser run level.
- Review and correct zone data errors – View load errors on the **Load Errors** page. Fix the errors and then repeat the previous tasks until no errors remain.
- Load data into operational zone tables – Set the server to the maintenance run level. Load the data from the staging tables into operational tables by using ClaimCenter web services or command prompt table import tools. Set the server to the multiuser run level.

See also

- *Administration Guide*

Files for zone data import

Importing zone data into staging tables uses CSV files that contain data. A data file is specific to an individual country or region. Configuration files define the columns that the data files provide.

Zone data files supplied by Guidewire

ClaimCenter provides a collection of zone data files for various localities with small sets of zone data that you can load for development and testing purposes. The zone data files are in the following location in the **Project** window in Guidewire Studio™.

configuration > config > geodata

In the **geodata** folder, ClaimCenter stores the zone information in country-specific **zone-config.xml** files, with each file in its own specific country folder. For example, the **zone-config.xml** file that configures address-related information in Australia is in the following location in the Studio **Project** window.

configuration > config > geodata > AU

Guidewire provides the **US-Locations.txt** and similar files for testing purposes to support autofill and autocomplete when users enter addresses. This data is provided on an as-is basis regarding data content. The provided zone data files are not complete and might not include recent changes.

Also, the formatting of individual data items in these files might not conform to your internal standards or the standards of third-party vendors that you use. For example, the names of streets and cities are formatted with mixed case letters but your standards may require all upper case letters.

The **US-Locations.txt** file contains information that does not conform to United States Postal Service (USPS) standards for bulk mailings. You can edit the **US-Locations.txt** file to conform to your particular address standards, and then import that version of the file.

Zone data configuration files

ClaimCenter provides sample zone data configuration files in subfolders of the following location in the Studio **Project** window.

configuration > config > geodata

These files specify the zone data columns, their order in the CSV data files, and the relationships among the zone data columns. To configure the address fields that you use and the structure of your addresses, edit these configuration files before importing zone data. For example, the configuration file for the US specifies that each line in the CSV data file must use the following format.

```
postalcode,state,city,county
```

Data that follows the format is shown below.

```
94114,CA,San Francisco,San Francisco
```

See also

- *Globalization Guide*

System parameters for database import

Guidewire provides an XML file, `database-config.xml`, containing parameters that support integration with the ClaimCenter database configuration. The `<loader>` element provides parameters that affect the use of staging tables to load data. For example, the `num-threads-integrity-checking` parameter supports multi-threading of data integrity checks. The `drop-deferrable-indexes` parameter controls whether to postpone the creation of performance-only indexes until after data is loaded. Some parameters are specific to the type of database that your ClaimCenter uses. For example, an Oracle database supports parallelism for multiple operation types.

You access `database-config.xml` in Guidewire Studio, in the following location.

configuration > config

You can view, but not edit, many of the database configuration parameters from within ClaimCenter at **Server Tools > Info Pages > Database Parameters**.

See also

- *Administration Guide*

Database import tables and columns

Importing data into the ClaimCenter database transfers data from an external system through staging tables into operational tables. Specific columns in these tables provide information about the import process. Other tables provide information about the results of integrity checks and the status of the import. ClaimCenter displays this information in info pages in the server tools user interface.

Never interact with any component of the ClaimCenter database directly, other than staging tables. Instead, use ClaimCenter APIs to abstract access to entity data, including all data model changes. Using APIs removes the need to understand many details of the application logic governing data integrity. The staging tables are exceptions because their purpose is to receive data that you provide from an external source.

Do not directly read or write the load error tables or the exclusion tables. The only supported access to these tables is the **Server Tools** user interface.

See also

- *Administration Guide*

Staging tables

Staging tables are database tables that replicate the loadable columns of corresponding operational tables. You prepare external data for bulk import into ClaimCenter operational tables by first loading the data into staging tables.

ClaimCenter creates staging tables when the server starts. ClaimCenter provides a staging table for any entity defined as `loadable` with at least one property also defined as `loadable`.

Not all operational tables have a corresponding staging table. For example, ClaimCenter does not use staging tables to import group hierarchies, roles, and system parameters. ClaimCenter does not provide staging tables for internal tables such as those for the message Send Queue, the internal SMTP email queue, and statistics information.

In the base configuration of ClaimCenter, all loadable operational tables have names with a `cc_` prefix. The corresponding staging tables have names with a `ccst_` prefix. For example, the operational table for the `Activity` entity is `cc_activity`. This table has a corresponding staging table, `ccst_activity`. Loadable data model extensions have operational tables with names that have a `ccx_` prefix. The corresponding staging tables for the data model extensions have names with a `ccst_` prefix.

A logical unit of work (LUW) groups related records across multiple staging tables into a single unit. All rows in an LUW must load from staging tables into operational tables. An error in any row in an LUW invalidates the whole LUW.

For example, the related Policy record must load into the ccst_policy staging table at the same time that a Claim record loads into the ccst_claim staging table. If the claim fails an integrity check, the policy also fails.

After staging table data loads successfully into operational tables, ClaimCenter deletes all the rows from the staging tables.

Load error table

The load error table holds data from failed data integrity checks. Do not directly read or write load error table data. Examine these tables using the **Server Tools > Info Pages > Load Errors** interface. Most errors relate to a particular staging table row, so the **Load Errors** page shows the following information.

- Table
- Row number
- Logical unit of work ID – The zone import tool creates the LUWIDs for zone data.
- Error message
- Data integrity check, also called the query, that failed

To resolve an error for an individual row, you can exclude the whole logical unit of work (LUW) from the import. Alternatively, use your conversion tool to correct the data. Other types of errors affect rows that ClaimCenter cannot identify by an LUWID. For example, some invalid ClaimCenter financials cause these types of errors. Resolving these types of errors requires additional investigation of the cause.

See also

- *Administration Guide*

Exclusion table

The load exclusion table contains the logical unit of work IDs (LUWIDs) to exclude from staging table processing during the next integrity check or load request. Do not directly read from or write to these tables. Use web services or command prompt tools to populate exclusion tables from LUWIDs in the load error tables. Use the same tools to delete the staging table rows defined by the LUWIDs in the exclusion table.

Load history table

The load history table stores results for import processes, including rows for each integrity check, each step of the integrity check, and row counts for the expected results. Use the information in these tables to verify that the import tools loaded the correct amount of data. You can view this information in ClaimCenter at **Server Tools > Info Pages > Load History**.

See also

- *Administration Guide*

Load command IDs

Running a staging table import copies all loadable entities from the staging tables to the operational tables. The staging table import run generates a load command id value. ClaimCenter sets the load command ID property (`LoadCommandID`) to this value on every entity that the load command imports. The command prompt tools that perform a database table import return the `LoadCommandID`. You can see the load command ID and track load import history using the user interface at **Server Tools > Info Pages > Load History**.

An entity's `LoadCommandID` property is always `null` for rows that did not enter ClaimCenter through staging table import. For example, a row that a user created by using the ClaimCenter user interface has a `null` value for the `LoadCommandID` property.

The `LoadCommandID` property does not change if the values of other properties on the entity change. If the user, application logic, or integration APIs change the data, the `LoadCommandID` property stays the same as when the row was first created.

Use the value of the `LoadCommandID` property to determine whether an entity was loaded using database staging tables or was created in some other way. Test an entity's `LoadCommandID` from your business rules or from a Java plugin. From Gosu, check `entity.LoadCommandID`. From Java, use the `entity.getLoadCommandID` method. The method returns a `TableImportResult` entity instance, which contains a `LoadCommandID` property, which is the load command ID. Call `result.getLoadCommandID()` to get the load command ID for that load request.

See also

- *Administration Guide*

Load user IDs

Running a staging table import populates user ID properties on imported entities. ClaimCenter sets the value of the `CreateUserID` and `UpdateUserID` properties to the user ID of the user that authenticated the web service or command prompt tool. To facilitate the selection of imported records by a database query, use a specific ClaimCenter user to import the staging table data into operational tables. For example, run the import commands as a user called Conversion. This user must have the SOAP Administration permission to execute the web service or command prompt tool. Do not give this user additional privileges or access to the user interface or other portions of ClaimCenter.

Use the value of the `CreateUserID` property to determine whether an entity was loaded using database staging tables or was created in some other way.

Load time stamps

Running a staging table import populates time stamp properties on imported entities. ClaimCenter sets the `CreateTime` and `UpdateTime` properties to the start time of the server transaction. All rows have the same time stamp for a single import run.

Import tools and commands

ClaimCenter provides both command prompt tools and web services APIs to support loading data from staging tables to operational tables in the database.

- Table import web service – The `TableImportAPI` web service provides table import functionality. This web service provides asynchronous methods, which return immediately and perform the command as a batch process. For example, `deleteExcludedRowsFromStagingTablesAsBatchProcess`. The web service methods return the identifier of the batch process that is running the command. Use the `MaintenanceToolsAPI` web service method `batchProcessStatusByID` to check for completion of the batch process.
- Table import command prompt tools – The `table_import` command prompt tool provides synchronous and asynchronous options to support importing data from staging tables into operational tables. To run the command asynchronously, add the `-batch` option. Asynchronous commands return immediately and perform the command as a batch process. For a list of options, run the following command from `ClaimCenter/admin/bin` and view the built-in help.

```
table_import -help
```

You must also write a tool to convert data from your source format and load it into the staging tables.

ClaimCenter also provides tools to load data from CSV files to the staging tables for zone data.

- Zone staging table import web service – The `ZoneImportAPI` web service provides functionality to import CSV data into the zone staging tables. All tools are provided as synchronous methods, which do not return until the command completes.

- Zone staging table import command prompt tools – The `zone_import` command prompt tool provides synchronous options to import CSV data into the zone staging tables. For a list of options, run the following command from `ClaimCenter/admin/bin` and view the built-in help.

```
zone_import -help
```

See also

- “Table import web service” on page 142
- “Zone data import web service” on page 146
- Administration Guide*

Server modes and run levels for database staging table import

The database import tools do not require the server to run in a specific mode. You can perform database staging table imports in any of production, test, or development modes.

For many of the tasks required to import data by using database staging tables and the database import tools, you must set the server to the maintenance run level. The maintenance run level prevents new user connections, halts existing user sessions, and releases all database resources. The server prohibits access from the ClaimCenter user interface to the database whenever the server runs at the maintenance level.

Example commands are shown below.

- API method: `SystemToolsAPI.setRunLevel(SystemRunlevel.GW_MAINTENANCE)`
- Command prompt tool: `system_tools -password password -maintenance`

See also

- “System tools web service” on page 140
- Administration Guide*

Database consistency checks

Database consistency checks ensure that the data in the ClaimCenter database is correct according to consistency rules. For example, the creation date for a row must always be earlier or the same as the update date. These checks are available from a command prompt tool, a web service, and the **Server Tools > Batch Processes** page. Database consistency checks always run asynchronously, as a batch process. The **Server Tools > Info Pages > Consistency Checks** page shows the results of running the consistency checks and the SQL commands that the checks run.

Consistency issues might take some time to resolve, so run consistency checks early in the data import project. Contact Guidewire Support for information on how to resolve consistency issues. Continue to run consistency checks periodically and resolve issues so that your database is ready when you begin the data import procedure.

Run these checks during the development of your conversion tool, using a test database, to ensure that the tool does not introduce errors into the database. Correct the errors in the conversion tool and rerun the tool until it produces no consistency errors.

Optionally, after loading the staging table data into the operational tables on the live ClaimCenter database, run consistency checks again.

Example commands are shown below.

- API method: `SystemToolsAPI.submitDBCCBatchJob()`
- Command prompt tool: `system_tools -passwordpassword -checkdbconsistency`

See also

- Administration Guide*

Your conversion tool

A critical component of any migration process is a conversion tool that you write. This tool converts your source data into ClaimCenter format and inserts the converted data into the staging tables. The conversion tool must map your source data into a format that contains the loadable columns of ClaimCenter operational tables. If your source format is dissimilar to the ClaimCenter format, this tool must support complex internal logic. The conversion tool groups sets of related records into logical units of work (LUW) and writes their IDs into staging table records. Loading data from the staging tables into the operational tables does not perform field-level validation, such as the format of an account number. The conversion tool must perform these types of checks.

Before loading data into the staging tables, clear existing data from the tables.

Example commands are shown below.

- API method: `var batchProcessID = TableImportAPI.clearStagingTablesAsBatchProcess()`
- Command prompt tool: `table_import -password password -clearstaging -batch`

Integrity checks

The ClaimCenter application includes a large set of database SQL queries that provide integrity checks on staging table data. These checks find and report problems that would cause the import to operational tables to fail or put ClaimCenter into an inconsistent state. Optionally, the integrity check command can clear error tables and exclusion tables. You can see the integrity check SQL queries at **Server Tools > Info Pages > Load Integrity Checks**. You can check if any integrity checks failed at **Server Tools > Info Pages > Load Errors**.

Before importing staging table data into operational database tables, ClaimCenter runs integrity checks. If the checks fail, ClaimCenter rolls back the data import. To avoid rolling back a data import, run the integrity checks and correct any errors before attempting the import. Even if the rows that caused errors in earlier integrity check runs were removed, ClaimCenter must rerun integrity checks before importing your data.

- If the logical unit of work IDs are populated incorrectly, the staging tables could contain extra rows that are not properly tied to an excluded claim.
- Identifying problems before a load starts is more efficient than triggering exceptions during the load process. If population of the operational tables encounters errors, the database must roll back the entire set of loaded records. Such rolling back of database changes is typically slow and resource-intensive.
- Some integrity check violations occur even if you remove all rows that contain errors from the staging tables. These violations occur for errors that cannot be tied to a single row.

The following descriptions contrast data integrity checks with other validations.

- The user interface (PCF) code enforces additional requirements.

For example, a property that is nullable in the database may require that users set a value in the ClaimCenter user interface. Importing a `null` value in this property passes integrity checks. However, if a user uses the ClaimCenter interface to edit an object containing the property, ClaimCenter requires a non-null value before saving because of data model validation.

- Integrity checks are different from validation rule sets and the validation plugin.

Integrity checks do not use field validators. If you need to ensure that a field fits a certain pattern, include that logic in your conversion tool before creating the value in the staging tables. For example, your conversion tool must validate that a claim number has the correct format.

Your conversion tool can trigger the integrity check after converting and loading the data into staging tables.

Before you run the checks, you must set the server run level to maintenance. After the checks complete, set the server run level back to multiuser.

Example commands are shown below.

- Web service method example

```
var batchProcessID =
    TableImportAPI.integrityCheckStagingTableContentsAsBatchProcess(clearErrorTable : true,
                                                                populateExclusionTable : false,
                                                                allowRefsToExistingNonAdminRows: false,
                                                                numThreadsIntegrityChecking : 8)
```

- Command prompt tool example

```
table_import -password password -integritycheck -clearerror -numthreadsintegritychecking 8 -batch
```

Types of integrity checks

The following partial list of data integrity checks shows the types of items that ClaimCenter enforces.

- No duplicate PublicID strings within the staging tables or in the corresponding operational tables
- No unmatched foreign keys
- No missing, required foreign keys. For example: an exposure that is not tied to a claim .
- No invalid codes for type key properties
- No invalid subtypes. For example: BI is not a valid exposure subtype.
- No null values in properties that must be non-null in operational tables and that do not provide a default value. Empty strings and text containing only space characters are treated as null values in data integrity checks for non-nullable properties.
- No duplicate values for any unique indexes. For example: ClaimNumber on cc_claim.

References to existing operational rows

By default, integrity checks allow references from data in staging tables to operational data in administrative tables only. For example, you can use references to existing users or groups, but you cannot load additional exposures or notes to an existing claim.

An optional parameter allows references to existing non-admin rows in the database. For the command-line tool, use the flag, `-allreferencesallowed`. For the web service, set the `allowRefsToExistingNonAdminRows` parameter to `true`. This parameter can cause performance degradation during the check and load process. Use this parameter only if absolutely necessary. For example, use this parameter in the rare case that a policy period overlaps between the existing operational data and the data you are loading.

See also

- [Administration Guide](#)

Database performance considerations

Guidewire strongly recommends that you design database import code to isolate your code performance from ClaimCenter database import API performance.

- For an Oracle database, take instrumentation snapshots (AWR snapshots or statspack snapshots at level 10) before and after `integritycheck` commands and `integritycheckandload` commands.
- Generate database instrumentation reports and resolve any performance problems. For an Oracle database, these reports are AWR reports or statspack reports. For SQL Server, the reports are DMV snapshots. Navigate to **Server Tools > Info Pages**. For an Oracle database, go to the **Oracle AWR** or **Oracle Statspack** page. For SQL Server, go to the **SQL Server DMV Snapshot** page. Download load information from the **Load History** page. Use all of this information to investigate performance problems in integrity checks and data loading.
- Navigate to **Server Tools > Info Pages > Database Statistics**. Download and archive a copy of the database catalog statistics before each database import attempt. Designing appropriate statistics collection into all database import code substantially improves the ability of Guidewire to advise you on performance issues related to database import.

Example commands are shown below.

- Web service method example

```
var batchProcessID = TableImportAPI.updateStatisticsOnStagingTablesAsBatchProcess()
```

- Command prompt tool example

```
table_import -password password -updatedatabestatistics -batch
```

Loading zone data into staging tables

Before you load zone data into staging tables, set up the zone data configuration files and the CSV files that contain the zone data. You must use a separate configuration file and CSV file for each country for which you need to load zone data. Before loading data into the staging tables, clear existing data from the tables.

Guidewire recommends that you load a complete set of zone data for a country rather than loading incremental changes to zone data. Incremental changes cause a non-linear growth in the time to import zone data and in the size of the zone link table. To reload data for a country for which zone data already exists, you must first clear the zone data for that country from the operational zone tables.

Example commands are shown below.

- Web service method examples

```
ZoneImportAPI.importToStaging("DE", "myzonedata.csv", clearStaging : true)  
ZoneImportAPI.clearProductionTables("DE")
```

- Command prompt tool examples

```
zone_import -import myzonedata.csv -country DE -clearstaging -server http://myserver:8080/pc \  
-user myusername -password mypassword  
zone_import -password password -clearproduction -country DE
```

See also

- “Zone data import web service” on page 146
- *Administration Guide*

Clearing errors from staging tables

The data rows in staging tables must not contain errors that cause integrity checks to fail. You can correct the errors by changing your conversion tool and reloading the data to the staging tables. Alternatively, you remove the logical units of work (LUW) that contain errors from the staging tables. Removing these LUWs is a two-stage process. First, you run integrity checks with the options to clear any previous error records and populate an exclusion table. Then, you delete the rows defined in the exclusion table from the staging tables.

Example commands to delete all rows from the staging tables are shown below.

- Web service method example

```
var batchProcessID = TableImportAPI.clearStagingTablesAsBatchProcess()
```

- Command prompt tool example

```
table_import -password password -clearstaging
```

Example commands to delete all rows from the zone staging tables are shown below.

- Web service method example

```
ZoneImportAPI.clearStagingTables()
```

- Command prompt tool example

```
zone_import -password password -clearstaging
```

Example commands to delete all rows from the error table are shown below.

- Web service method example

```
var batchProcessID = TableImportAPI.clearErrorTableAsBatchProcess()
```

- Command prompt tool example

```
table_import -password password -clearerror
```

Example commands that populate the exclusion table with all of the distinct logical unit of work IDs that are in the error table are shown below.

- Web service method example

```
var batchProcessID = TableImportAPI.populateExclusionTableAsBatchProcess()
```

- Command prompt tool example

```
table_import -password password -populateexclusion -batch
```

Example commands that delete all rows from all staging tables that match a logical unit of work ID in the exclusion table are shown below.

- Web service method example

```
var batchProcessID = TableImportAPI.deleteExcludedRowsFromStagingTablesAsBatchProcess()
```

- Command prompt tool example

```
table_import -password password -deleteexcluded -batch
```

Example commands to delete all rows from the exclusion table are shown below.

- Web service method example

```
var batchProcessID = TableImportAPI.clearExclusionTableAsBatchProcess()
```

- Command prompt tool example

```
table_import -password password -clearexclusion
```

Importing data into operational tables

Importing data into operational tables always runs integrity checks on the data in the staging tables before doing the import. The staging tables must contain no rows that trigger an integrity check error. If an integrity check fails, ClaimCenter rolls back the whole import.

Before you import data into operational tables, you must set the server run level to maintenance. After the import completes, set the server run level back to multiuser.

The server removes all data from staging tables on successful completion of the load. If the integrity check generated errors, data remains in the staging tables.

The load command accepts the same optional `allowRefsToExistingNonAdminRows` parameter to allow references to existing non-admin rows in the database that the integrity check command does. This parameter can cause performance degradation during the check and load process. Use this parameter only if absolutely necessary. For example, use this parameter in the rare case that a policy period overlaps between the existing operational data and the data you are loading.

Example commands are shown below.

- Web service method examples

```
var batchProcessID =
    TableImportAPI.integrityCheckStagingTableContentsAndLoadSourceTablesAsBatchProcess(
        clearErrorTable : true,
        populateExclusionTable : false,
        updateDBStatisticsWithEstimates : false,
        allowRefsToExistingNonAdminRows : false,
        numThreadsIntegrityChecking : 1)
```

- Command prompt tool example

```
table_import -password password -integritycheckandload -batch
```

Automated setting of properties and entities

The ClaimCenter load process uses callbacks to create various properties and entities. These callbacks populate user and time properties on imported entities as well as other internal or calculated values. If a property has a null value in the staging table and the data model specifies a default value, ClaimCenter sets the value of that property to the default value. In the data model files, properties that the callbacks create have the `loadedByCallback` attribute set to true.

ClaimCenter sets the value of the `CreateUserID` and `UpdateUserID` properties to the user ID of the user that authenticated the web service or command prompt tool. To facilitate the selection of imported records by a database query, use a specific ClaimCenter user to import the staging table data into operational tables.

Importing zone data into operational tables

To import only zone data to the operational tables, use the `-zonedataonly` option on the command prompt tool. For the web service `TableImportAPI`, call the method

`integrityCheckZoneStagingTableContentsAndLoadZoneSourceTablesAsBatchProcess`. Specifying zone data import runs only the integrity checks for zone data and therefore completes in less time.

Oracle database statistics

The Oracle database optimizer chooses a query plan based on the database statistics. If you load large amounts of data to operational tables that have no records or only a few records, the tables in the database grow significantly. To ensure that the optimizer uses a suitable query plan after the load completes, Guidewire recommends that you update the database statistics to reflect the expected size of the table.

The load tools provide an option to estimate the database statistics for row and block counts when you perform the load. Use of this option causes the load tool to execute row counts and set database statistics on the operational tables based on the contents of the staging tables. For the web service, set the `updateDBStatisticsWithEstimates` parameter to true. For the command prompt tool, use the `-estimateorastats` option. For a non-Oracle database, ClaimCenter ignores this parameter and option.

Estimate the database statistics only on Oracle databases that contain few or no claim records. Estimating the statistics on databases that have more data significantly reduces database performance.

Using a conversion tool to populate the staging tables

The first step in importing data using the staging tables is mapping external data to the ClaimCenter staging tables. There is a one-to-one correspondence between staging tables and operational tables. The **Data Entities (Migration View)** link in the *Data Dictionary* shows the tables and columns that load from the staging tables into operational tables. This view shows the names of operational tables, which have a prefix of `cc_`. The corresponding staging tables have a prefix of `ccst_`.

For example, the staging table for the ClaimCenter `cc_claim` table is the `ccst_claim` table. Similarly, the staging table for the ContactManager `ab_address` table is the `abst_address` table.

The migration view shows only the tables and columns that ClaimCenter tools load from staging tables into operational tables. The *Data Dictionary* marks these items with the loadable flag. Other properties are either not backed by database columns or are not importable from staging tables. For example, the staging tables contain no virtual

properties, which are generated (calculated) from other properties at run time. Use the other sets of pages in the *Data Dictionary* to see the entity and database views that show the full sets of tables and columns.

For example, look at the `Claim` entity in the *Data Dictionary*. The property `AccidentType` shows the `(loadable)` flag, which indicates that the property is included in the staging table for claims.

The columns in staging tables differ in some ways from those in the operational tables. You can see these differences by using a database management tool to examine the columns.

- The staging table contains a row number column and a `LUWID` column that do not exist in the `cc_` tables.
- The following properties exist in operational tables but not in the staging tables: `CreateUserID`, `UpdateUserID`, `LoadCommandID`, `BeanVersion`, `CreateTime`, and `UpdateTime`.
- Staging tables do not include properties that track internal system state because ClaimCenter itself sets these properties at run time.
- Properties that contain typelist codes in operational tables, for example, `State`, have a data type of `String` that ClaimCenter cannot be certain of the value of the property upon import. Mark the claim's `state` as either open or closed upon import using the appropriate typelist codes.
- Operational tables contain a load command ID column, which indicates whether an entity was loaded from database staging tables.

Important properties and concepts for database import

Most business data properties are straightforward to populate in the staging tables. Review the *Data Dictionary* to understand the meaning of the loadable columns on tables. Put appropriate values into the columns of the same name on the staging tables. The following subsections provide general guidelines for how to populate the staging tables.

Public IDs and the retired property

The operational tables in the ClaimCenter database use a unique ID column as a primary key. The staging tables do not have a unique ID column. Staging tables use a public ID column to identify rows. Your conversion tool defines the public IDs. Typically, you use the primary key from the source data as the public ID. Alternatively, create a public ID value by concatenating values.

For example, the `PublicID` for each claim could be the company name, a colon character, and the claim number. The `PublicID` for exposures could be the public ID of the claim number, a colon character, and the claim revision order for the exposure as an incrementing counter.

Do not set a public ID to a string that starts with a two-character ID and then a colon because Guidewire reserves all such IDs. If you use the `PublicIDPrefix` configuration parameter to change the prefix of auto-created public IDs, conversion code that sets explicit public IDs must not conflict with that name space. Plan the public ID format carefully to ensure that your system can support a significant number of records and stay within the 64 character limit for a public ID.

For `Transaction` and `TransactionLineItem` entities, the character length limit for public IDs is 62, not 64.

The public ID column, `PublicID`, in database tables contains a string value that uniquely identifies a row in the table, in conjunction with the `Retired` column. The public ID value in a staging table must also be different from any public ID in the corresponding operational table. If multiple rows have the same public ID, the `Retired` columns must have unique values. A non-zero value for the `Retired` property indicates that the row does not contain active data. You can load a staging table row with a public ID that matches the public ID of a row in the related operational table. In this case, either the staging table row or the operational table row must have a non-zero value in the `Retired` property.

Guidewire recommends that you do not load retired data during staging table import.

Foreign keys

The operational tables in the ClaimCenter database use the primary key values of parent tables as foreign keys to link dependent items to the parent. For example, the `PrimaryAddress` column of a `Contact` record is a foreign key that links to the primary key of an `Address` record. In the staging tables, the foreign key columns have a `String` data type

and must have a value that is the Public ID of the target. Your conversion tool must ensure that these foreign key columns contain the correct Public ID values.

For example, the `ClaimID` property on `ccst_exposure` must contain the public ID (`publicID`) of the claim entity. To resolve the `ClaimID` foreign key, ClaimCenter looks in both the `ccst_claim` and `cc_claim` tables.

During the load phase, ClaimCenter looks up the foreign key by public ID and insert its internal ID into the operational table.

Logical units of work, LUW, and LUWIDs

A logical unit of work (LUW) is a set of related rows that must load into operational tables as a single unit. For example, related `Address` records must load into the `ccst_address` staging table at the same time that a `Contact` record loads into the `ccst_contact` staging table. These `Contact` and `Address` records make a logical unit of work. If either the `Contact` record or an `Address` record fails an integrity check, the whole set of records is invalid. Each LUW must have a unique identifier (LUWID). You must identify self-contained units of data that must load together or fail together if an integrity check fails on any part of the unit. Your conversion tool defines these LUWIDs and writes them into the `LUWID` column of the staging table. Use a meaningful value for an LUWID, such as the identifier for the primary record in the source data.

For example, for claim data, you typically use the claim number. If you insert users, you could set the LUWID to the user's login name (for example, `ssmith`) because the user is probably related to more than one claim. You can check whether the user already exists in ClaimCenter. If the user exists, you can use the LUWID value to locate and delete all the related staging table data for the user. The related data is in the `ccst_user`, `ccst_credential`, and `ccst_contact` tables.

The zone import tool creates an identifier (LUWID) for each LUW that it defines. For example, a `Zone` record for a postal code must load into the `ccst_zone` staging table at the same time as a `Zone` record for its city.

An LUWID is used in the following ways.

- To identify which data failed.
- To specify data to exclude from future integrity checks or load requests.

Nullable columns in the staging tables

Most columns in the staging tables are nullable, even if the corresponding column in the operational table is not nullable. This difference enables you to insert rows and set default values later, if necessary, while loading the staging tables.

If the corresponding column in the operational table has a default value, as shown in the *Data Dictionary*, you can leave the staging table column `null`. ClaimCenter sets the default value. If the property does not have a default value, you must set a value in the staging table. You can use your conversion tool to set a default value.

Encrypted columns

If some data entities contain encrypted properties, you must encrypt those values before importing the staging table data into operational tables. ClaimCenter provides a tool to encrypt properties in your staging tables.

Typekey columns

Operational tables in the ClaimCenter database use the typecode data type to enforce a restricted set of values in particular columns. The staging tables use the String data type for the equivalent columns. Your conversion tool must set the values in these columns to a String representation of a valid `Code` value from the typelist. During data load, ClaimCenter converts this code to an internal integer typecode ID.

For example, the `AccidentType` column on the `cc_claim` table must contain this type of code value.

Subtyped tables

If the operational table is subtyped, the staging table contains a subtype column. Use the text name of the subtype as the value of the column in the staging table. Look in the *Data Dictionary* for the subtype names.

For example, for cc_contact the subtypes include Company, Person and Company.

Row number columns

Do not write a value in the row number column, RowNumber. The row number property is used only for reporting errors. ClaimCenter populates this column on inserting the row or by the integrity check process, depending on which database management system you are using.

Overwritten tables and columns

For some operational tables, the staging tables are always overwritten during import. In the data model files, these entities have the overwrittenInStagingTable attribute set to true. Similarly, some properties have both the loadable and overwrittenInStagingTable attributes. Do not write any values to staging tables or columns that have the overwrittenInStagingTable attribute set to true. ClaimCenter uses these staging tables and columns during the load of staging table data to operational tables. The staging table loader subroutines overwrite any values that your conversion tool writes into these staging tables and columns.

Virtual properties

Some Guidewire entities contain properties called virtual properties. Gosu treats these properties similarly to regular properties, for example, when reading property values. However, the database does not store virtual properties as columns. If Gosu requests the value of a virtual property, the value is generated dynamically. In the base configuration of ClaimCenter, the staging tables do not contain loadable columns for virtual properties. Your conversion tool is responsible for populating only loadable properties and columns. In some cases the virtual property calls a method that performs a complex calculation that pulls values from many different properties or even other entities. To find out which properties are used to generate the virtual property, see the *ClaimCenter Data Dictionary*. For some virtual properties, the *Data Dictionary* contains information about the calculation of the value of the virtual property. Use the **Data Entities (Migration View)** link on the *Data Dictionary* to show only loadable entities and loadable columns.

Indexes in staging tables

The ClaimCenter database provides indexes on every staging table to ensure that integrity checks perform well.

Handling errors in database import using a conversion tool

During development of the conversion tool, you run integrity checks to identify data errors. Typically, you run many trial conversions and integrity checks with iterative algorithm changes before resolving all errors. In many cases, incorrect mapping of data from the source system into the staging tables causes integrity checks to fail. Errors also occur if you do not populate all required properties. Typically, you fix these errors by adjusting algorithms in your conversion tool.

A problem in the source data can cause an integrity check to flag an error. You can use SQL commands to correct the data directly in the staging tables. Never modify operational tables with direct SQL commands. Alternatively, you can correct errors in the source system. In that case, remove records that fail the integrity check from the staging tables. Correct the errors in the source system. Then, rerun the entire conversion process.

Loading ClaimCenter entities

To generate rows and correctly set necessary properties on staging tables in the ClaimCenter database, you must understand the data model. See the following section for more information restrictions on loading financial transactions. The following table describes the requirements for specific ClaimCenter entities.

Table	Column	Requirement
ccst_claim	State	Governs the claim's open or closed status. Set to open or closed for all claims being imported. If closed, also set the ClosedDate property.
	PolicyID	Every claim must have its own copy of policy information. Do not use the same policy ID value on multiple claims even if more than one claim is associated with the same policy revision. The policy row for a claim must contain a snapshot of the policy information at the time of the claim, specified by the LossDate column.
ccst_address ccst_contact ccst_contactaddress ccst_contactcatsScore ccst_contactcontact ccst_contacttag ccst_eftdata	AddressBookUID	Each claim in ClaimCenter uses its own copy of contact information. Integrate ContactManager with ClaimCenter to ensure that multiple copies of a person, company, or address have the same information. Use the AddressBookUID column of a shared record to link to the record in ContactManager. Set the column to one of the following values. <ul style="list-style-type: none"> • The LinkID field of a ContactManager record to link to an existing record in ClaimCenter. For example, for a contact, link to an ABContact entity. For an address, link to an Address entity. • null to make the record local to the claim and not appear in the central address book nor be available to other claims.
ccst_contact	AutoSync	To enable synchronization of all contacts that share a contact record in ContactManager, set the value of this column to the text value of AutoSync . Allow. If you use a different value for this column, the contact information on different claims for the same contact can differ.
ccst_transaction	Status	ClaimCenter only supports insertion of financial transactions that have already been processed. The status must have a post-submission state, such as submitted for a reserve.
	CheckID (payments only)	Every payment must be part of a check. If importing a payment from an external system, you must also create a row in the ccst_check table to hold check information, such as the check number. Set the value of this column to the public ID of the check.
	TransactionSetID	You can enter multiple transactions as part of a set that is approved as a batch, such as a set of reserve changes. All imported transactions must be part of a transaction set, even if there is only a single transaction in the set. You must create a row in the ccst_transactionset table to hold transaction set information. Set the value of this column to the public ID of the transaction set.

Requirements for importing policy data

You must load a Policy entity for each claim. For each Policy entity, you must load all the associated entities that the policy requires. For example, you must load the coverages for every policy. Some policy types require other entities, such as risk units, locations, or workers' compensation statistical codes.

Requirements for importing policy location data

The **Policy Location** page in the user interface displays policy location and building information from policy location (**PolicyLocation**) entities. ClaimCenter displays this information only if a related risk unit (**RiskUnit**) entity of subtype **LocationBasedRU** exists. Create the risk unit row in the staging table at the same time as you create the policy location and building data. If you do not load this data, ClaimCenter does not display the policy location information in the user interface.

Financial transaction importing requirements

ClaimCenter enforces certain requirements on the data that you import to avoid changing the meaning of existing data. For example, you cannot add a new **TransactionLineItem** to an existing **Transaction** or add another reserve to a reserve set. You must import entire financial occurrences as a unit.

ClaimCenter enforces these restrictions by preventing the following types of imports.

- New financials data for a claim that already has any financials data on it.
- Additional transactions into an existing transaction set. For example, do not add another reserve into an approved reserve set. Do not add another check or reserve into a check set. Do not add another recovery into a recovery set. Do not add another recovery reserve into a recovery reserve set.
- Additional payments into an existing check.
- Additional deductions into an existing check.
- An additional `TransactionLineItem` into an existing `Transaction`.

Do not import the following types of records for financials data.

- Transactions and checks that have not yet been processed. No events would be generated for these checks, so they would never get processed.
- Recoded payments. Use the user interface to recode payments. For example, checks are recoded by adding onset and offset payment transactions that change the categorization of the payments.
- Transferred checks. Use the user interface to transfer checks after import completes.
- Multi-payee checks. Use the user interface to handle multi-payee checks after import completes.
- `TransactionSet` records that are not approved.

You must also follow the following rules for financials data.

- For supplemental payments, you must ensure that you do not violate the rules of the system. For example, if the configuration parameter `AllowNoPriorPaymentSupplement` is set to `false`, do not load supplemental payments on claims that have no other financial transactions. Additionally, never load supplemental payments into open claims or exposures.
- Financial entities of types `Reserve`, `Recovery Reserve`, or `Recovery` must have the status `submitted`.
- A `Payment` entity must have the status `submitted`, `voided`, or `stopped`.
- A `Check` entity must have the status `requested`, `issued`, `cleared`, `voided`, or `stopped`.
- A `Check` entity must have at least one payee.
- If the `AllowMultipleLineItems` configuration parameter is set to `false`, each `Transaction` can have only one `TransactionLineItem`.
- If the `AllowMultiplePayments` configuration parameter is set to `false`, each `Check` can have only one `Payment`.
- The `PublicID` property of a transaction must be no longer than 18 characters, which is two characters fewer than the maximum property length.

The following rules apply to multicurrency support on transactions.

- If you use the web service API to import transactions, you can pass a set of exchange rates. ClaimCenter creates the complete set of all permutations of `ExchangeRate` and `ExchangeRateSet` entities for those rates. ClaimCenter then links `Transaction` entities in each `TransactionSet` to the rates.
- If you import multicurrency transactions using staging tables, you must ensure that every `Transaction` entity references a custom `ExchangeRate` entity. Every multicurrency `Transaction` imported through staging tables has a custom exchange rate. Multicurrency payment transactions on the same check can share the same custom `ExchangeRateSet` entity. `Reserve`, `Recovery`, and `RecoveryReserve` multicurrency transactions must each point to their own custom `ExchangeRate` entity.

The following integrity checks run for transactions that are imported through staging tables.

- The payments in a check all point to the same `ExchangeRate` entity.
- Check payments have all `null` or all non-null transaction to claim exchange rates.
- The sign of the transaction amount, whether positive or negative, matches the sign of the claim amount.

- The transaction to claim exchange rate has the correct currencies. Specifically, the ExchangeRate entity has its transaction currency as ExchangeRate.BaseCurrency and the claim/default currency is ExchangeRate.PriceCurrency.
- All check payments have the same currency.

Batch processing requirements for database import

During development of your data conversion process, run the following types of batch processing to ensure data integrity.

- Database Consistency Checks
- Bulk Claim Validation

Run this batch process after performing data import from staging tables to operational tables to ensure the quality of the loaded claim data.

If you use additional ClaimCenter features, you might want to run the following batch processes.

- Aggregate Limit Calculations
- Claim Health Calculations

The following batch processes are optional, but Guidewire highly recommends that you run them.

- Claim Contacts Calculations

Run this batch process if your conversion process does not set the values of denormalized properties such as Exposure.ClaimantDenorm. Guidewire recommends that your conversion tool sets the values of these denormalized properties. Setting these values avoids the delay of running this batch process after importing staging table data before ClaimCenter can go live.

- Data Distribution

Run this batch process to ensure the quality of the loaded claim data.

- Dashboard Statistics

In the base configuration, this batch process runs daily at 1:00 a.m. Run immediately after data import to provide statistics that include the loaded data.

See also

- *Administration Guide*

Detailed work flow for a typical database staging table import

Plan carefully when to perform a database import on production systems. Users are blocked from using the application during that time. A database import operation can take considerable time, depending on how much data you want to import. To reduce the effect of down time for users, consider importing large amounts of data in phases rather than in a single operation.

You must write your conversion tool and test it in a development environment before performing the tasks in this section.

Migrating data from a source or legacy system to ClaimCenter typically happens in the steps described in the subsequent sections.

Prepare the data and the ClaimCenter database

You must perform multiple tasks before you load data from staging tables.

Procedure

1. If you changed the data model, run the server to perform upgrades. The staging table import tools require that source data be converted to the same format as the server data.

2. Determine which records to migrate. Your conversion tool determines which source records to migrate according to your own criteria. For example, the tool generates an extraction list of claim numbers for claims to convert.
3. Create zone data files. Optionally, create zone data files in comma separated value (CSV) format.
4. Set the server to the maintenance run level. Set each ClaimCenter server to the maintenance run level by using the `system_tools` command prompt tool or the web service `SystemToolsAPI.setRunLevel` method.
5. Back up the operational tables in the database. Back up operational tables before import.
6. Clear the staging tables, the error tables, and the exclusion tables. Your conversion tool typically does this task. Alternatively, you can use web service APIs or command-line tools.
7. Run ClaimCenter database consistency checks. Database consistency checks verify that the data in your operational tables does not contain any data integrity errors. Run database consistency checks using the web services `SystemToolsAPI.checkDatabaseConsistency` method, or running the command prompt tool as shown below.

```
system_tools -password password -checkdbconsistency
```

Import data into the staging tables

You must ensure that the data that you import into the staging tables does not cause errors in integrity checks.

About this task

Repeat the following steps until the integrity checks report no errors.

Procedure

1. Convert and insert source data into staging tables. Your conversion tool reads the claims to convert from the source system and converts their data to ClaimCenter format. Then, your tool inserts the converted data for each claim into the staging tables, along with associated objects in related tables. This step represents the bulk of your effort.
2. Run the zone import tool. If you are loading zone data, run the command prompt tool `zone_import` or use the `IZoneImportAPI` web service to import your CSV data.
3. Set the server to the maintenance run level. Set each ClaimCenter server to the maintenance run level by using the `system_tools` command prompt tool or the web service `SystemToolsAPI.setRunLevel` method.
4. Request integrity checks from the table import tools. Use the `TableImportAPI` web service method `integrityCheckStagingTableContentsAsBatchProcess` or the following command prompt command.

```
table_import -password password -integritycheck -batch
```

For an Oracle database, take instrumentation snapshots before and after running integrity checks. Generate database instrumentation reports to identify performance problems.

Note: The `-batch` option does not wait until the started process completes before returning. Instead, it returns immediately and prints the ID of the started process (PID). The process caller is responsible for waiting for the process to complete before taking further action.

5. Set the server to the multiuser run level. Set each ClaimCenter server to the multiuser run level by using the `system_tools` command prompt tool or the web service `SystemToolsAPI.setRunLevel` method.
6. Check load errors from the Load History page. This page is available at **Server Tools > Info Pages > Load History**.
7. Resolve any errors. If there are load errors, you must correct the faulty data rows or exclude them from the import.

Either the conversion tool must correct the errors or you can exclude the bad data. The user interface displays the rows that caused the integrity checks to fail. Use those rows to make a redo list, which is a list of records to fix and rerun. Alternatively, use this data to find records to skip by adding to the exclusion table. To exclude records that cause errors, add LUWIDs to the exclusion table by using the web service API or command prompt tools.

Then, delete rows from those LUWIDs from the staging tables. Some errors are not associated with specific LUWIDs, in which case nothing can be moved to the exclusion table.

For example, some types of financials errors are not associated with specific LUWIDs.

- a) Decide whether to correct the errors or exclude the rows from the import. Go to **Server Tools > Info Pages > Load Errors**. This page displays the records that contain errors. Use the list of rows to decide which rows to correct and which to exclude from the import.
 - b) Fix errors in the source data or revise your conversion tool code. For rows that you want to correct, you have two choices. Either correct the data in the source or revise the conversion tool to load those rows correctly.
 - c) Fix errors in the zone data source files. If you are loading zone data, edit the zone data CSV file to correct the data.
8. Decide whether to populate or clear the error and exclusion tables. The error table contains information about the rows that failed the earlier integrity checks. If you want to clear these errors before the next integrity checks run or exclude data rows that cause errors, perform the following steps.
- a) Exclude rows that caused integrity check errors. Populate the load exclusion table with the LUWIDs that contain errors. Use the `TableImportAPI` web service method `populateExclusionTableAsBatchProcess` or the `table_import` command prompt tool.
 - b) Remove excluded records from staging tables. Remove all rows from all staging tables for records whose LUWIDs are listed in the exclusion table. You can use the `table_import` command prompt tool or the `TableImportAPI` web service method `deleteExcludedRowsFromStagingTablesAsBatchProcess`.
 - c) Clear the error and exclusion tables. Optionally, for clarity, remove the errors and exclusion rows related to earlier integrity checks. Use the `TableImportAPI` web service method `clearErrorTableAsBatchProcess` or the `table_import` command prompt tool. When you next run integrity checks, only the errors from that run appear in the user interface.

Handling encrypted properties

If you need to use staging tables to import records and any data has encrypted properties, you must encrypt those properties before importing them into operational tables. The table import tools provide options to encrypt any encrypted fields in staging tables before import. These tools encrypt only the columns that are marked as encrypted in the data model.

During upgrades, be careful about the order in which you make data model modifications. Before beginning staging table import work, do any data model changes including changing encryption settings. Next, let the server upgrade the database. After you modify the data model, during server launch, ClaimCenter automatically encrypts data in the operational tables as part of upgrade routines. During upgrade, ClaimCenter deletes all staging tables.

Guidewire strongly recommends that you encrypt your staging table data after you complete all work on your data conversion tools and your data passes all integrity checks. Note that staging table integrity checks never depend on the encryption status of encrypted properties. You can run integrity checks independent of whether the encrypted properties are currently encrypted.

ClaimCenter provides the following ways to encrypt your data.

- Asynchronously with the command line tool `table_import` with the options `-encryptstagingtbls -batch`.
- Asynchronously with web services using the `TableImportAPI` web service method `encryptDataOnStagingTablesAsBatchProcess`. The method takes no arguments and returns a process ID. Use the returned ID to check the status of the encryption batch process or to terminate the encryption batch process. Use the ID with methods on the web service `MaintenanceToolsAPI`.

See also

- “Encrypt any encrypted properties in staging tables” on page 272

Load staging table data into operational tables

After there are no errors in integrity checks, you can load data into operational tables from staging tables.

Before you begin

Perform these tasks after integrity checks succeed for all records except for logical units of work (LUW) records in the exclusion table. Then, delete the LUWs that relate to records in the exclusion table. At this point, you can load data from the staging tables into the operational tables used by ClaimCenter.

Procedure

1. Update the database statistics for the staging tables. Use the `TableImportAPI` web service method `updateStatisticsOnStagingTablesAsBatchProcess` or the `table_import` command prompt tool. For example, run the following command at a command prompt.

```
table_import -password password -updatedatabestatistics -batch
```

Note: The `-batch` option does not wait until the started process completes before returning. Instead, it returns immediately and prints the ID of the started process (PID). The process caller is responsible for waiting for the process to complete before taking further action.

2. In a cluster installation, shut down all ClaimCenter servers except one. Use the `system_tools` command prompt tool or the web service `SystemToolsAPI.scheduleShutdown` method.
3. Set the one running ClaimCenter server to the maintenance run level. Use the `system_tools` command prompt tool or the web service `SystemToolsAPI.setRunLevel` method.
4. Load staging table data into operational tables. Use the `TableImportAPI` web service method `integrityCheckStagingTableContentsAndLoadSourceTablesAsBatchProcess` or the `table_import` command prompt tool with the `-integritycheckandload` option. ClaimCenter inserts data rows into the operational tables and results information into the load history table.

- For an Oracle database, take instrumentation snapshots before and after running the integrity check and load. Generate database instrumentation reports to identify performance problems.
- For an Oracle database, if the operational tables contain no records or few records, use the `load` option to estimate database statistics, as shown in the following command.

```
table_import -password password -integritycheckandload -estimateorastats -batch
```

- For a non-Oracle database or an Oracle database that already contains many records, use the following command.

```
table_import -password password -integritycheckandload -batch
```

- To load only zone table data into operational tables, set the `zone data only` option. For an Oracle database that has few records, to load zone data only, also set the option to estimate database statistics, as shown in the following command.

```
table_import -integritycheckandload -estimateorastats -zonedataonly
```

- For a non-Oracle database or an Oracle database that already contains many records, to load only zone table data into operational tables, use the following command.

```
table_import -password password -integritycheckandload -zonedataonly
```

5. Set the server to the multiuser run level. Set each ClaimCenter server to the multiuser run level by using the `system_tools` command prompt tool or the web service `SystemToolsAPI.setRunLevel` method.

Perform post-import tasks

After you load data into operational tables from staging tables, you must perform additional tasks to ensure that ClaimCenter and its database perform correctly.

Procedure

1. Review the load history at **Server Tools > Info Pages > Load History**.
2. Ensure that the amount of data loaded is correct.
3. Update database statistics. Particularly after a large conversion, update database statistics.
4. Optionally, run database consistency checks. New consistency errors after importing data indicate that the staging table data contained errors that integrity checks did not detect.
5. If necessary, import additional records into the staging tables.
6. Run the required batch processes.

Table import tips and troubleshooting

The following items describe important information concerning importing data using the staging tables.

- The staging table import commands that perform integrity checks require the server to be at the `maintenance` run level. This run level prevents users from logging in to the ClaimCenter application.
- ClaimCenter runs the load process inside a single database transaction to be able to roll back if errors occur. The load process may require a large rollback space. Run the import in smaller batches, for example, a few thousand records at time, if you are running out of rollback space.
- As with any major database change, make a backup of your database prior to running a major import.
- After loading staging tables, update database statistics on the staging tables. Update database statistics on all the operational tables after a successful import. After a successful import, ClaimCenter provides table-specific update database statistics commands in the `cc_loaddbstatistics` command table. You can selectively update statistics on only the tables that actually had data imported, rather than for all tables.
- Always run database consistency checks on the operational database tables both before and after table imports. If there were no consistency errors before importing, new consistency errors after importing data indicate that the staging table data contained errors that integrity checks did not detect.

For example, inconsistency can occur with financials totals that are denormalized and stored as totals for performance reasons. ClaimCenter does not validate that the denormalized amount in the staging tables is correct. It would be extremely slow to do so. To correct this type of problem after loading the data, recalculate denormalized values by using the following tool.

```
maintenance_tools -password password -startprocess financialscalculations
```

- During ClaimCenter upgrade, the ClaimCenter upgrader tool may drop and recreate staging tables. This activity occurs for any staging table for which the corresponding operational table changed and requires upgrade. Never rely on data in the staging tables remaining across an upgrade. Never perform a database upgrade during your import process.

FNOL mapper

The FNOL mapper is a ClaimCenter integration tool that imports First Notice of Loss (FNOL), or initial claim, reports. Typically, the FNOL mapper imports FNOL report data from XML documents in the standard XML format known as ACORD XML for Property & Casualty. However, you can configure the FNOL mapper to import report data in other custom XML formats.

- For ACORD XML data, the external system calls the `ClaimAPI` web service method `importAcordClaimFromXML`.
- For custom XML data, the external system calls the `ClaimAPI` web service method `importClaimFromXML`.

With either web service method, the external system passes XML data to ClaimCenter as one large `String` object. With both methods, the external system calls the `ClaimAPI` web service once for each FNOL report. External systems cannot pass a batch of reports in a single call. The `importClaimFromXML` method has an additional parameter for the special custom mapper class that you must write to handle the custom format.

See also

- For more information on the ACORD XML for Property & Casualty standard, visit the following website.
<https://www.acord.org/standards/downloads/Pages/default.aspx>

Importing ACORD XML data with the FNOL mapper

With the FNOL mapper tool, you can import FNOL reports in the ACORD XML for Property & Casualty format. ACORD XML is a standard XML format for FNOL reports, but variants of the format and claim and exposure data models vary widely. It is impossible to provide a default mapping that works for all cases.

For example, if reports have additional properties on exposures, you must specify how to map the additional properties to the ClaimCenter data model. The names of properties or typecodes might vary between the reports and ClaimCenter. If you have custom exposure types, you might need to change the exposure type based on other fields in the reports.

Custom mapping of ACORD XML data

To customize your ACORD XML mapping, modify the built-in Gosu classes or write new ones to handle each report object specially. The included classes are modular already to keep exposure mapping code separate from contact mapping code and from address mapping code.

Special handling of ACORD XML data

The XSD for ACORD XML allows multiple FNOL reports to be included in one XML document. However, the FNOL mapper tool requires that ACORD XML documents contain a single report. External systems must call either method on the `ClaimAPI` web service once for each FNOL report.

An ACORD XML document can have many <MClaimSvcRq> subelements. Each subelement can have many <CLAIMSSVCRQMGs> subelements with many <xsd:CHOICE> subelements. One of the <xsd:CHOICE> subelements must be a <ClaimsNotificationAddRq> element. The default implementation of the FNOL mapper uses the first valid <ClaimsNotificationAddRq> element in the XML document. If the document has no valid <ClaimsNotificationAddRq> elements, Gosu throws an exception.

Importing custom XML data with the FNOL mapper

With the FNOL mapper tool, you can import FNOL reports in non-ACORD XML formats. To import from custom XML formats, write new Gosu classes that implement special FNOL mapper interfaces. The mapper interfaces indicate that your classes can map the incoming XML.

FNOL mapper detailed flow

Of all the server files for the FNOL mapper tool, the most important top-level component is the interface `FNOLMapper` in the `gw.api.fnolmapper` package. The `FNOLMapper` interface defines the contract between ClaimCenter and a class that can map XML for FNOL reports to a `Claim` entity and its subobjects. The `FNOLMapper` interface has a single method, `populateClaim`.

Another important component of the FNOL mapper tool is the `ClaimAPI` web service interface. For ACORD XML data, external systems call the interface method `importAcordClaimFromXML`. For a custom mapper class for non-ACORD XML data, external systems call the method `importClaimFromXML`. The `importClaimFromXML` method takes an extra parameter for the name of your custom mapper class.

You can invoke the FNOL mapper for ACORD XML data from Java web services client code, such as in the following statement.

```
String myClaimPublicID = claimAPI.importAcordClaimFromXML(myXMLData);
```

For both methods on the `ClaimAPI` web service interface, the web service implementation itself is relatively simple because the FNOL mapper classes do most of the work.

The general flow of the FNOL mapper web service is described below.

1. Find the appropriate implementation class for the mapper and instantiate it. This class must implement the `FNOLMapper` interface. For the `importAcordClaimFromXML` method, ClaimCenter always uses the built-in mapper implementation class. For the `importClaimFromXML` method, the second argument is the name of your custom mapper implementation class.
2. Create a new `Claim` entity.
3. Create a new `Policy` entity and attach it to the new claim.

Call the `populateClaim` method on the mapper class and pass the new `Claim` entity. The method signature is shown below. The mapper must throw `FNOLMapperException` if it cannot map the claim.

```
function populateClaim(Claim claim, String xml) : void
```

4. The mapper class reads the XML data to set fields on the `Claim` entity and add subobjects as appropriate.
5. After the `populateClaim` method completes, the web service persists the new claim and its subobjects.
6. The web service returns the public ID (a `String` value) for the imported and persisted claim. If errors occur, the web service API throws an exception to the web service client.

Structure of FNOL mapper classes

To support basic ACORD files, ClaimCenter provides in its base configuration an implementation of the `gw.api.fnolmapper.FNOLMapper` interface. This implementation is the Gosu class `gw.fnolmapper.acord.AcordFNOLMapper`. The class maps a basic ACORD file starting at the

<ClaimsNotificationAddRq> subelement to the ClaimCenter base configuration data model for claims and exposures, including typecodes. A real-world implementation likely requires some customization to one or more of these files.

You can use the default implementation and modify it directly. However, the AcordFNOLMapper class delegates most of its important work to other mapping interfaces to handle specific element objects. The following table lists the component, the interface class that manages this mapping, and the name of the class that implements the default behavior.

Data to map	Interface to implement to manage the mapping	Provided plugin implementation class
Address	gw.fnolmapper.acord.IAddressMapper	gw.fnolmapper.acord.impl.AcordAddressMapper
Contact	gw.fnolmapper.acord.IContactMapper	gw.fnolmapper.acord.impl.AcordContactMapper
Exposure	gw.fnolmapper.acord.IExposureMapper	gw.fnolmapper.acord.impl.AcordExposureMapper
Incident	gw.fnolmapper.acord.IIncidentMapper	gw.fnolmapper.acord.impl.AcordIncidentMapper
Policy	gw.fnolmapper.acord.IPolicyMapper	gw.fnolmapper.acord.impl.AcordPolicyMapper

To make significant changes to the default mappings, you could either modify the base configuration implementation classes or you could create new implementations of the interfaces. Each of these interfaces provides a method for each variant of this object that is possible in the ACORD XSD.

For example, for contacts, the ACORD format supports two contact types, `InsuredOrPrincipal_Type` and `ClaimsParty_Type`. In ClaimCenter, both contact types correspond to `ClaimContact` entities. The `IContactMapper` interface defines a method for each of the ACORD types. The method in the mapper must be able to convert the ACORD XML encoding of that type into a ClaimCenter `ClaimContact` object, which it returns. The `AcordContactMapper` class implements this interface. However, you could provide a different implementation.

Using Gosu native XML support with FNOL mappers

From Gosu, the mappers do not manipulate the XML as raw text. Gosu includes native XML support and the ability to read and write a Gosu representation of XML data as nodes as native Gosu objects. In cases where an XSD is available, which is the case for ACORD, properties on node objects have the correct compile-time type from Gosu. For example, a claim party type in the ACORD spec appears in Gosu as an XML node object as the type `xsd.acord.ClaimsParty_Type`. From Gosu you can work with these objects naturally in a type-safe way. You do not need any direct parsing of XML as text data.

Instantiating address, contact, exposure, or incident mapping classes

To instantiate the implementation classes for the specialized mappers for addresses, contacts, exposures, and incidents, the ACORD mapper calls the `AcordMapperFactory` class, which implements the `IMapperFactory` interface.

If you want to replace the implementation for any of the ACORD-specific mapping classes (for address, contact, exposure, or incident), perform one and only one of the following operations.

- Modify the simple built-in `AcordMapperFactory` class to instantiate your own versions of mapping classes.
- Alternatively, provide a new `IMapperFactory` implementation class based on the default one. However, if you do this you must modify `AcordFNOLMapper` to instantiate your new mapper factory class instead of the default mapper factory. However, in the built-in implementation of the ACORD mapper, the root `AcordFNOLMapper` class does not

directly create the mapper factory. Instead, it delegates this work to the configuration utility class `AcordConfig`. Modify that file and look for the following code statements.

```
protected function createMapperFactory() : IMapperFactory {
    return new AcordMapperFactory(this)
}
```

Change the statements to instantiate your own mapper factory as shown below.

```
protected function createMapperFactory() : IMapperFactory {
    return new abc.claimcenter.fnolmapper.ABCMapperFactory(this)
}
```

Configuration files and typecode mapping for ACORD data

ClaimCenter includes a utility class that supports storing mapping configuration information in a configuration directory and accesses that information with convenience methods.

The core configuration file class is `gw.api.fnolmapper.FNOLMapperConfig`. You can use that class with your own mappers.

For the ACORD implementation in the base configuration, ClaimCenter includes the configuration utility class, `gw.fnolmapper.acord.AcordConfig`. This class gets an instance of the `FNOLMapperConfig` class and uses its methods and properties. The base configuration ACORD implementation of FNOL mapping, `gw.fnolmapper.acord.AcordFNOLMapper`, calls `AcordConfig` methods directly.

Using the mapper properties file for ACORD data

The `mapper.properties` file in the `ClaimCenter/modules/configuration/config/fnolmapper/acord/` folder specifies the following basic properties.

`mapper.alias.default`

A default alias name that is used if no mapping is found between an external code and a Guidewire typekey name in the mapping XML file. If you do not specify this property, there is no default alias, in which case, Gosu throws an exception eventually for unmapped typecodes. ClaimCenter returns this exception to the web services client as a SOAP fault.

In the base configuration, the value of this property is `default`.

`mapper.file`

The path of the XML mapping file for mapping Guidewire typecodes to codes that the XML to be mapped uses.

In the base configuration, the value of this property is `typecodemapping.xml`, which refers to the file in the following location.

```
ClaimCenter/modules/configuration/config/fnolmapper/acord/
```

If you comment out this property, the mapper uses the following file.

```
ClaimCenter/modules/configuration/config/typelists/mapping/typecodemapping.xml
```

To modify the mapper properties for ACORD data, edit the following file.

```
ClaimCenter/modules/configuration/config/fnolmapper/acord/mapper.properties
```

You can add more properties to this file.

You get properties from this file by using properties and methods on the class `gw.api.fnolmapper.FNOLMapperConfig`, including the following.

`getDefaultKey()` method

The value of the `defaultKey` property as a `String`. From Gosu, you can use the `DefaultKey` property instead of calling this method.

`getFile(filename) method`

A method that returns a file for the given String file name as a File object.

`getLogger() method`

The logger used to record messages in the mapper. From Gosu, you can use the Logger property instead of calling this method.

`getProperties(name) method`

Returns a Properties object from a properties file with the given name. Do not include the .properties extension on the name parameter.

`getTypecodeMapper() method`

The value of the typecode mapper property. The return type is gw.api.util.TypecodeMapper. From Gosu, you can use the TypecodeMapper property instead of calling this method.

Typecode mapping for ACORD data

Although you can use a TypecodeMapper object to map typecodes given a typelist, a namespace, and an alias, the methods that it provides manipulate the typekeys as String values. This approach is not type-safe. ClaimCenter provides the Gosu class TypeKeyMap as a type-safe wrapper around TypecodeMapper. You can use TypeKeyMap for performing alias-to-typekey conversions for a specific typelist.

The AcordConfig class defines a series of TypeKeyMap getter methods for each typelist. The base configuration uses these methods to map enumerations in the ACORD XML to Guidewire typekeys in a type-safe and readable way.

For example, the AcordConfig class defines a method called `getContactRoleMap` to map contact roles. This returns a map of ContactRole typecodes so you can map the external alias to the internal code.

The contact mapping class AcordContactMapper uses the typecode map to convert a role name to the internal typecode for contact roles with easy-to-read Gosu code.

```
private function getRole(roleName:String) : ClaimContactRole {  
    var claimRole = new ClaimContactRole()  
    claimRole.Role = config.getContactRoleMap().get(roleName)  
    return claimRole  
}
```

If you write or customize your own mappers, consider following this approach for readability.

Additional classes and enhancements for FNOL mapping

The following classes and enhancements are used for FNOL mapping.

ACORD utility class with constant definitions

The included simple class AcordUtil contains simple constants for processing ACORD XML, such as role IDs. Refer to the implementation in Studio for details.

ContactManager class in Guidewire ContactManager

In the base configuration, the ContactManager class tracks the roles and contacts on the current claim. For detailed information on how this class is implemented, open the class in Guidewire Studio.

ACORD XSD type enhancements

ClaimCenter includes special Gosu enhancements for dates, times, and addresses to make it easier to manipulate ACORD XSD types. For details, refer to the source for the following Gosu enhancement files.

DateEnhancement

Enhances the type xsd.acord.Date to add two methods `toDate` and `toDateTime` to convert it to a `java.util.Date` object. The `toDateTime` version takes an argument, which is the time of day in a `gw.xml.xsd.types.XSDTime` object.

DateTimeEnhancement

Enhances the type `xsd.acord.DateTime` to add a `toDate` method to convert it to a `java.util.Date` object.

DetailAddressEnhancement

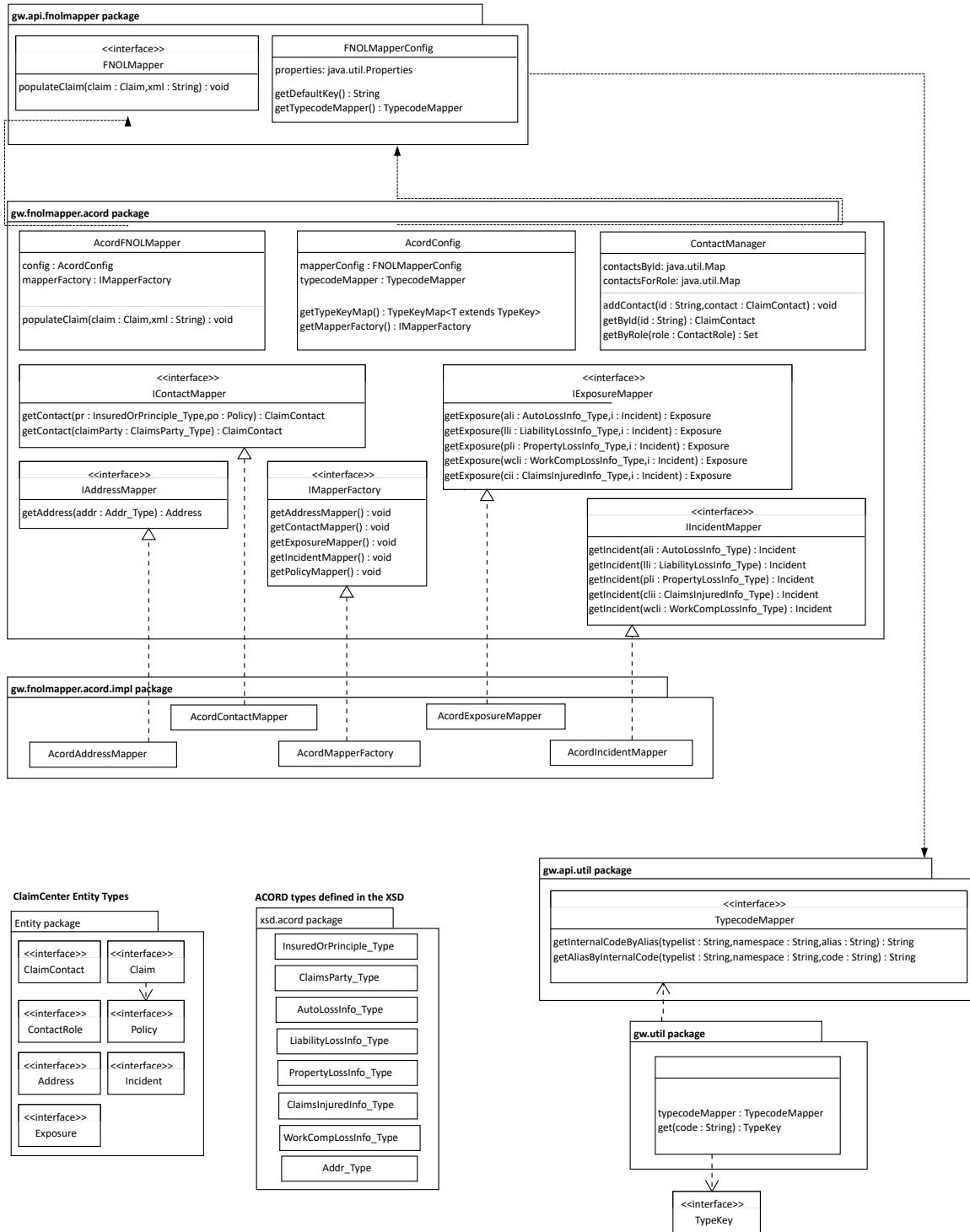
Enhances the type `xsd.acord.DetailAddr_Type` to add a `DisplayName` property that contains the display name.

ClaimsPartyEnhancement

Enhances the type `xsd.acord.ClaimsParty_Type` to add the `RoleCodes` property to return role codes (`ClaimsPartyRoleCd`) for this `ClaimsParty`.

FNOL mapper and built-in ACORD class diagram

The following diagram shows the relationships of different classes in the built-in ACORD mapper.



Example FNOL mapper customizations

Add mapping for a new exposure type

To use a new exposure type, you write classes to define how to map an incident to that exposure type.

About this task

Suppose you want to map a general liability incident to one of two different exposure types in ClaimCenter according to custom properties in the ACORD file. To add mappings for a new exposure type, you follow steps similar to the following.

Procedure

1. In Studio, create a new implementation of the `IExposureMapper` interface based on the code in the built-in `AcordExposureMapper.gs` file. For this example, we assume your company is called ABC and you name your class `ABCExposureMapper.gs` in the package `abc.claimcenter.fnolmapper`.
2. In Studio, create a new implementation of the `IMapperFactory` interface based on the code in the built-in `AcordMapperFactory.gs` file. For this example, we assume your company is called ABC and you name your class `ABCMapperFactory.gs` in the package `abc.claimcenter.fnolmapper`.
3. Modify class `ABCMapperFactory.gs` to create your own exposure mapper.

```
override function getExposureMapper(contactManager:ContactManager) : IExposureMapper {
    return new abc.claimcenter.fnolmapper.ABCExposureMapper(acordConfig, contactManager)
}
```

4. In Studio, modify the built-in classes so that the `FNOLMapper` class calls your new mapper factory. In the built-in implementation of the ACORD mapper, the root `AcordFNOLMapper` class does not directly create the mapper factory. Instead, it delegates this work to the configuration utility class `AcordConfig`.
 - a) Look for the following code statements.

```
protected function createMapperFactory() : IMapperFactory {
    return new AcordMapperFactory(this)
}
```

- b) Change the statements to match the following.

```
protected function createMapperFactory() : IMapperFactory {
    return new abc.claimcenter.fnolmapper.ABCMapperFactory(this)
}
```

5. In your `ABCExposureMapper` class, locate the method that maps the `PropertyLossInfo_Type` XML object.

```
//Return an exposure for a General Liability incident
override function getExposure(c:Claim, glLossInfo:LiabilityLossInfo_Type,
    incident:Incident) : Exposure {
    var exposure = new Exposure()
    exposure.ExposureType = ExposureType.TC_GENERALDAMAGE
    exposure.PrimaryCoverage = CoverageType.TC_GL
    exposure.Incident = incident
    exposure.LossParty = LossPartyType.TC_THIRD_PARTY
    return exposure
}
```

6. Change the line that assigns a value to `exposure.ExposureType`. Set the value based on other fields in the `glLossInfo` object.

The first argument to all `getExposure` method variants is a reference to the claim. Use that reference to access properties and objects that are already mapped by previously run FNOL mapper code. For example, suppose you already mapped a ClaimCenter `Incident` entity. Code that runs later can access the newly mapped incident. For example, if you want to get or set the vehicle incident property `VehicleIncident.Driver`, you need to reference the incident.

Add additional properties to an exposure

To use a new exposure type, you write a class to define how to map the exposure properties to properties in the ClaimCenter data model.

Before you begin

Suppose that for general liability incidents in the ACORD file, you need to set additional data model extensions in ClaimCenter based on custom properties in the ACORD file.

First, follow the steps in “Add mapping for a new exposure type” on page 574 except for the final step.

Procedure

- Instead of that final step, in your ABCExposureMapper class, modify the method that maps the PropertyLossInfo_Type XML object.

```
// Returns an exposure for a General Liability incident
override function getExposure(c:Claim, glLossInfo:LiabilityLossInfo_Type,
    incident:Incident) : Exposure {
    var exposure = new Exposure()
    exposure.ExposureType = ExposureType.TC_GENERALDAMAGE
    exposure.PrimaryCoverage = CoverageType.TC_GL
    exposure.Incident = incident
    exposure.LossParty = LossPartyType.TC_THIRD_PARTY
    return exposure
}
```

- Before the return statement at the end of that method, set the values of the additional properties based on other fields in the glLossInfo object.

The first argument to all getExposure method variants is a reference to the claim. Use that link to access properties and objects that are already mapped by previous-run FNOL mapper code. For example, suppose you already mapped a ClaimCenter Incident entity. Code that runs later can access the new mapped incident. For example, if wanted to get or set the vehicle incident property VehicleIncident.Driver, you need to reference the incident.

Create a new mapper for non-ACORD data

In some cases, new exposure types do not use the ACORD data format. For these exposure types, you write a class to define how to map the exposure properties to properties in the ClaimCenter data model.

About this task

Suppose you need to use the basic FNOL Mapper structure to map XML data for an FNOL, but the data is not in the ACORD format. You perform the following steps.

Procedure

- Create a new implementation of the FNOLMapper interface. For this example, we assume your company is called ABC and you name your class ABCFNOLMapper.gs in the package abc.claimcenter.fnolmapper.
- To invoke the mapper, instead of your client code calling the ClaimAPI web service interface method importAcordClaimFromXML, instead call importClaimFromXML. The importClaimFromXML method takes an extra parameter for the name of the mapper class you want to use.

```
String myClaimPublicID;
myClaimPublicID = claimAPI.importClaimFromXML(myXMLData, "abc.claimcenter.fnolmapper.ABCFNOLMapper");
```

- Your ABCFNOLMapper.gs file does not need to follow the pattern of the built-in ACORD mapper. However, you can review the structure of the ACORD mapper for useful patterns you can duplicate.
 - Encapsulate your mapping code for addresses, contacts, exposures, incidents, and policies into separate files.

- b)** You can add Gosu enhancements on XSD-specific types. Use these enhancements to encapsulate the implementation of type-specific code and make your mapping code high-level and easy to read.
- c)** Use the configuration files and the TypeKeyMap utility class, as used in the ACORD mapper.

ISO and Metropolitan

ClaimCenter supports integration with the following industry services:

- The ClaimSearch service that helps detect duplicate and fraudulent insurance claims that the insurance service organization ISO provides.
- The nationwide police accident and incident report service in the United States that the Metropolitan Reporting Bureau provides.

chapter 28

Insurance services office (ISO) integration

The insurance service organization ISO provides a service called ClaimSearch that helps detect duplicate and fraudulent insurance claims. If an insurance company enters a claim, the company can send details to the ISO ClaimSearch service, and get reports of potentially similar claims from other companies.

The full ISO XML data hierarchy is in the ISO-provided documentation the *ISO XML User Manual* with file name `XML User Manual.doc`. ISO strictly enforces this hierarchy. Always request the latest version of this manual from ISO and refer to it during any customizations. If you want to send ISO optional properties that ClaimCenter does not send by default, use the *ISO XML User Manual* to find the properties.

For more information, refer to the following ISO ClaimSearch website.

<https://www.verisk.com/insurance/products/claimsearch/>

ISO integration overview

ClaimSearch helps detect duplicate and fraudulent insurance claims. After an adjuster enters a claim, the insurer can send details to the ISO ClaimSearch service and get reports about potentially similar claims from other companies. ClaimCenter includes built-in integration with this service.

After an insured customer calls an insurance company with a claim, the insurance company sends details to the ISO ClaimSearch service to save the claim's basic information. ISO returns a response with reports describing potential duplicate claims. ISO might also send match reports later if another company enters a matching claim into their database. Consequently, a single submission to ISO may result in several responses: the initial matches and then additional matches, possibly much later.

ClaimCenter requires use of the ISO ClaimSearch for DataPower Platform service, which is the newer version of the ISO ClaimSearch service. Contact ISO to sign up for this service. You will need to know your public key for the SSL certificate of the inbound callback ISO URL.

Claim-based messaging with legacy support for exposures

For new claims, always send claims, not exposures, to ISO. For compatibility with older versions of ClaimCenter that supported sending exposures, ClaimCenter supports receiving updates from ISO for claims with previously-reported exposures. Similarly, ClaimCenter can send exposure updates for claims with previously-reported exposures.

If you previously used exposure-level ISO integration, ISO tracks your previous submissions of ClaimCenter exposures in the ISO system. ClaimCenter handles match reports that ISO sends later based on exposures submitted

before the switch to claim-based ISO messaging. ClaimCenter continues to maintain the following ISO-related properties on the exposures.

ISOSendDate

The ISO send date, which is set in the same database transaction as the new message to ISO.

KnownToISO

The Boolean flag that indicates whether ISO has acknowledged the message. This property is set in the message acknowledgment database transaction.

If ClaimCenter attempted to send an exposure to ISO (the `ISOSendDate` is non-null) ClaimCenter handles ISO replies on that claim by using only exposure-based mode. After a user attempts to send ISO a claim or exposure, that claim is fixed in that mode, claim-based or exposure-based. This behavior allows a smooth transition from exposure-based ISO messaging to claim-based ISO messaging while still handling exposures submitted to ISO before the switch to claim-based messaging.

The actual implementation of this behavior is a property called `ISOCClaimLevelMessaging` on claims. This Gosu enhancement property returns `true` to indicate that ClaimCenter uses claim-level messaging for this claim. It returns `false` to indicate exposure-based messaging.

Configure claim-based messaging

About this task

In the default configuration, claim-level ISO reporting does not report a claim to ISO until all exposures reach the ISO validation level. You can change this behavior to cause ClaimCenter to report claims when one or more exposures reach ISO validation level. Perform the following steps to send a claim when one or more exposures reach ISO validation level.

Procedure

1. Edit the `ISO.gs` file.
2. In the `checkForClaimChanges` method, look for the following method call.

```
claim.isValid("iso", true)
```

The second argument indicates whether to check the validation levels of all exposures before proceeding. If you pass the value `true`, ClaimCenter skips ISO reporting if any exposure has not yet reached ISO validation level. Instead, change the second argument to `false`.

```
claim.isValid("iso", false)
```

3. Add a filter to remove exposures that are not yet at ISO validation level. To begin, edit the file `ISOCClaimSearchRequestBase.gs`.
4. Find the `createClaimLevelSearchPayload` method which iterates through exposures while inserting them into the ISO Claims Search payload.
5. Add a filter to remove non-ISO-ready exposures.

```
for (exposure in Claim.ISOOrderedExposures) {
    if (exposure.isValid(ValidationLevel.TC_ISO)) {
        addExposureToClaimLevelSearchRequest(exposure)
    }
}
```

Configuring key fields

Key fields are a set of fields in ClaimCenter that, when changed, automatically trigger sending messages to ISO. Key fields are defined in the `keyFieldChanged()` method in `ISO.gs`, as shown in the following:

```
static function keyFieldChanged(reportable : ISOResponsible) : boolean {
    var claim = reportable.ISOClaim
```

```
        return claim.isFieldChanged("ClaimNumber")
           || claim.isFieldChanged("AgencyId")
           || claim.Policy.PolicyNumber != claim.OriginalPolicyNumber
           || (claim.isFieldChanged("LossDate")
               && DateUtil.compareIgnoreTime(Coercions.makeDateFrom(claim.getOriginalValue("LossDate")), claim.LossDate) != 0)
    }
```

Note that changes to the LossDate field are compared to the time of day the loss occurred. If the loss time has changed, but not the date, ClaimCenter does not send claim data to ISO.

Add additional fields to the `keyFieldChanged()` method using `isFieldChanged()` or other logic to trigger ClaimCenter sending data to ISO.

Match reports

`ISOMatchReport` is a delegate that defines the interface for ISO match reports. The claim and exposure versions of ISO match reports implement this interface. The `ClaimISOMatchReport` and `ExposureISOMatchReport` entities contain the properties `ISOClaim` and `ISOExposure`. For `ClaimISOMatchReport`, the `ISOClaim` property points to the claim and the `ISOExposure` property is null. For `ExposureISOMatchReport`, the `ISOExposure` property points to the exposure and the `ISOClaim` property points to the claim that owns that exposure.

If you use exposure-based messaging, remember that the difference in terminology between ClaimCenter and ISO. For exposure-based messaging, ISO refers to a “claim,” while ClaimCenter calls the same item an “exposure.”

For new claims, always send claims not exposures to ISO. Exposure-based messaging is only supported for claims with previously-reported exposures by earlier versions of ClaimCenter.

ClaimCenter has a special validation level for ISO. When a claim or exposure is ready to send to ISO, the validation level matches or exceeds this level. Once an exposure reaches this level, ClaimCenter sends the claim to ISO and records any ISO match reports associated with the exposure.

The relative position of the levels is defined by the `Priority` field on each validation level.

Implementation overview

ClaimCenter ISO integration components include the following components.

- **ISO user interface in ClaimCenter** – The built-in user interface includes the following elements.
 - A button to manually send/resend a claim or exposure to ISO
 - A tab in the exposure details page for users to view ISO information on an exposure or claim
 - Pages to view the details of a particular ISO match
- **ISO validation level and related validation rules** – Once validation rules indicate a claim or exposure reached the ISO validation level, ClaimCenter knows the claim or exposure can be sent to ISO.
- **ISO messaging destination and related event rules** – After a claim or exposure reaches the ISO validation level, or if it changes after reaching this level, built-in messaging rules detect the change. The rules send a request to ISO built-in messaging destinations to generate messages to ISO. ClaimCenter includes built-in Event Fired rules written in Gosu that assemble the payload to send to ISO. The largest part of real-world ISO integrations are customizing these payload-generation rules for changes to your data model and special business requirements, such as custom exposure types.

ClaimCenter provides payload generating Gosu rules for the most common payload types, and sends all required properties for those payloads. To generate different ISO payloads, to add ISO optional properties, or to add new properties that ISO requires, customize your payload generation functions. The built-in rules provide nearly all of what you probably need for initial deployment, not including your data model changes. You must modify the Gosu code to match your data model changes, to generate new types of payloads, or to add ISO optional properties unsent in ClaimCenter by default.

- **ISO receive servlet that waits for responses from ISO** – After the servlet gets an ISO response, it notifies the destination and updates the claim or exposure with the new response and match report. A match report from ISO describes data from other insurance companies about an exposure that seems similar to a claim or exposure in your

system. ClaimCenter stores match reports as ClaimCenter documents. The most important details are also added to a new array of `ISOMatchReport` subtype entities within the ClaimCenter claim or exposure.

- **ISO properties on ClaimCenter claims, exposures, and vehicles** – For ISO support, ClaimCenter stores special properties on `Claim`, `Exposure` and `Vehicle` entities, plus certain ISO-specific typelists. Additionally, ClaimCenter stores ISO responses as match reports on the claim or exposure. Additionally, for incoming match reports, ClaimCenter generates documents on the claim or exposure as linked `Document` entities.
- **ISO properties files** – There are additional configuration options in the ISO property files.
- **ISO mapping files** – There are additional configuration options in the mapping files for coverages and typecodes.
- **Administration pages to examine outgoing messages** – Although the Administration pages for messages are not ISO-specific, outgoing requests to ISO are handled by using the messaging system. It is sometimes necessary for ClaimCenter administrators to track connectivity issues by using this user interface.

See also

- “ISO user interface” on page 602
- “ISO validation level” on page 598
- “ISO messaging destination” on page 599
- “ISO payload XML customization” on page 608
- “ISO receive servlet and the ISO reply plugin” on page 601
- “ISO match reports” on page 610
- “ISO properties on entities” on page 601
- “ISO properties file” on page 603
- “ISO type code and coverage mapping” on page 607

Reference material for ISO integration work

The full ISO XML data hierarchy is in the ISO-provided documentation the *ISO XML User Manual* with file name is `XML User Manual.doc`. ISO strictly enforces this hierarchy. Always request the latest version of this manual from ISO and refer to it during any customizations. If you want to send ISO optional properties not sent by ClaimCenter by default, use the ISO XML User Manual for the reference of all their properties.

For additional related information, refer to the ISO ClaimSearch web site.

<https://www.verisk.com/insurance/products/claimsearch/>

What to know about ISO mapping

The most important thing to know about ISO integration is that largest part of real-world ISO integrations are customizing payload-generation rules. You probably need to customize the rules for changes to your data model and special business requirements, such as custom exposure types. Additionally, you probably need to change typelists and update ISO payload rules accordingly. This process is called *mapping* the ClaimCenter view of your business data with ISO’s structure for similar data.

Secondly, you need to know that mapping is very important and that business analysts are critical for every ISO integration team to help with mapping.

It is critical that before ISO integration work begins, business analysts determine the relationship between custom exposure types and typecode values and their corresponding types in ISO. This information is used in mapping files that map ClaimCenter typecodes to their ISO equivalents as well as in the payload generation rule customizations.

You must allocate time in your integration projects to determining these relationships, customizing the ISO integration, and testing it. Additionally, even after you have pushed your system to production, if you make ongoing changes to your data model, you must make matching ISO changes. For example, if you add new typecodes to a typelist used by ISO integration, you must update mapping files that map the ClaimCenter new typecodes to the ISO equivalents.

Mapping is typically involves some estimation and research, and sometimes there is no clear answer to a mapping issue. For example, by default, vehicle damage in ClaimCenter maps to what ISO calls a property payload. For a variety of business reasons, this mapping tends to lead to better ISO match result quality, even though it tends to lower the number of total returned matches. Give yourself adequate time for designing, finalizing, and testing your mapping. Allow at least three to five weeks.

You might additionally need to carefully consider how any user interface changes in ClaimCenter might affect your required properties. For example, it may seem like hiding some properties on exposures or subobjects you do not use would have no negative effects. However, if those properties are strictly required by ISO, hiding those properties in the user interface is typically not recommended due to the changes required. If you choose to do it anyway, you must make changes in the payload generation rules to add placeholder data in the XML payload. Making these changes causes ISO to accept the records as valid so that it does not return errors.

Set your expectations that you need more ISO integration mapping work every time you make any user interface changes on exposures or any of its subobjects. More work is especially required if you hide built-in properties. As a general rule, if you make changes after initial ISO deployment, you may require additional ISO-related mapping work.

Similarly, if you modify the data model on any typelists on ISO required properties, you might need to rewrite typecode mapping and possibly the payload generation rules. Set your expectations that you might need more ISO integration mapping work every time you make data model changes on exposures or any of its subobjects. If you make changes after initial ISO deployment, you may require additional ISO-related mapping work.

The mapping process is typically not extremely complex. However, it is time consuming and typically is an ongoing process because of ongoing data model and user interface changes. You must adapt the mapping process for your own business needs.

The most important files related to mapping are listed below.

- The ISO payload generation Gosu rules, especially to accommodate new exposure types
- The ISO properties files, especially to map settings related to the `ClaimContact` entity. If you want more complex claim contact mapping than supported by the ISO property files, you can customize the Gosu that loads this properties file.
- The ISO typecode mapping file `TypeCodeMap.xml`
- The ISO coverage mapping file `ISOCoverageCodeMap.csv`

Implementation technical overview of ClaimSearch web service

ISO ClaimSearch is a web service accessed by using the SOAP protocol over a secure sockets (SSL) connection. ISO provides two ClaimSearch systems: the live system and a test system. Both systems provide the same interface and functionality, but the test system allows clients to test and experiment with their ISO integrations without affecting the live system.

Each system provides two operations: `SubmitToISO` and `SubmitTestToISO`. Both take a single argument which is an XML string. The XML is the ACORD format with a few ISO extensions. The XML data describes the claim submitted to ISO or updates to a claim already known to ISO. The following table compares and contrasts the two operations.

Operation	Behavior
<code>SubmitToISO</code>	<code>SubmitToISO</code> checks the XML and queues it up for addition to the ISO database. After the request processes, ISO sends an asynchronous response.
<code>SubmitTestToISO</code>	<code>SubmitTestToISO</code> checks the XML but does not add it to the queue. The ISO database does not update and ISO does not send an asynchronous response. <code>SubmitTestToISO</code> is only useful for testing that you can connect to ISO and submit correctly formatted XML.

At the time you register as a new customer with ISO, you must specify a range of IP addresses from which you send requests. You must also specify a callback URL to which ISO sends asynchronous responses. ISO allocates a customer ID and password that you must put in the header of every XML request. All communication to and from ISO is done over secure sockets (SSL) so your ID and password cannot be intercepted.

You submit ISO requests with the following steps.

1. ClaimCenter submits a request by using `SubmitToISO`.
2. ISO receives a request and immediately checks all of the following criteria.
 - The request contains a valid customer ID and password.
 - The source IP address of the request is within the range allocated to the customer.
 - The request contains correctly formatted XML.
3. ISO sends back the return value, which is a short XML message known as a receipt. If the ClaimCenter request passes the initial checks, then the receipt contains the status value `ResponsePending`. This status value means that the request queued up and will process soon. If the request did not pass the checks, then the receipt contains an error message.
A good receipt does not guarantee that the request is correct. ISO performs most of its detailed error checking as it actually processes the request later.
4. ISO processes the request. First, the request is checked for errors. For instance, all required properties must be present. Also, policy, coverage, and loss types must be consistent. If the request is correct, it is added to or updated in the ISO database. Then, a query is run to find if there are other matching claims.
5. ISO sends a response to your callback URL. If the request was valid, then the response contains a `Success` status value plus details of any matches. If the request was invalid, then the response contains a description of the problem.
6. At a later time, if another company submits a claim to ISO and it matches one you submitted, then ISO sends another response to your callback URL. The response contains the new list of matching claims.

There are some variations to this basic protocol. ClaimCenter requests to ISO fall into three major categories: adding a new claim, replacing the information about an existing claim, and updating an existing claim. In most cases, you can specify with a property in the XML data, whether or not you want a search to be done after the add/replace/update.

Enable ISO integration

Before you begin

The ISO messaging destination handles outgoing requests to ISO, and it includes the following components.

- A message transport plugin that handles outgoing ISO communication
- A message reply plugin that handles incoming ISO messages

To enable ISO support, you must enable the ISO messaging plugin and the ISO messaging transport.

Procedure

1. In Project window of Guidewire Studio, navigate to **configuration > config > Messaging**, and then open the file `messaging-config.xml`.
2. In the list on the left, click the row for ID 66 and the name `Java.MessageDestination.ISO.Name`.
3. Select the **Enabled** check box.
4. Navigate to **configuration > config > Plugins > registry**, and then `isoTransport.gwp`.
5. Select the **Enabled** check box.

What to do next

If you need to disable the ISO support at a later time, follow the same procedure but deselect the **Enabled** checkbox in both editors.

ISO implementation checklist

Integrating Guidewire ClaimCenter with ISO ClaimSearch requires some planning and interactions with ISO. Guidewire recommends that project teams that plan ISO integration must include the deployment steps described in this topic in project planning timelines.

1. “ISO membership” on page 585
2. “Team resources” on page 585
3. “Requesting ISO testing and production server access” on page 585
4. “ISO configuration and integration testing” on page 586
5. “ISO integration data migration” on page 586
6. “ISO production testing, including connectivity testing” on page 586
7. “ISO production deployment” on page 587

ISO membership

If your company is new to the Insurance Services Office, you must purchase an ISO membership.

Purchasing a membership typically includes contract negotiation with ISO. This can take various lengths of time depending on the needs of both companies, corporate legal interaction, and final steps for approving and finalizing contracts. Guidewire recommends that projects begin this membership phase immediately to reduce delays to later ISO integration phases.

Once membership is complete, ISO provides you with an ISO username, an ISO password, and ISO company code (an ISO-specific identification code separate from the ISO username).

Be sure you get contact information for people at ISO for both administrative and technical contacts.

See also

- “ISO authentication and security” on page 596

Team resources

You must ensure you have the appropriate set of team members for an ISO integration. The team must fill the following positions.

- **A business analyst from the insurance company for mapping coverages and properties** – ClaimCenter provides payload generating Gosu rules for the most common payload types, and sends all required properties for those payloads. If you want to generate different ISO payloads, or add ISO optional properties, or add properties that ISO later makes required, customize the payload generation functions. The largest part of real-world ISO integrations are customizing these payload-generation rules for changes to your data model and special business requirements, such as custom exposure types. In practice, a business analyst from the insurance company is the best resource for understanding the business requirements and the coverage implications for their business needs.
- **Engineers to implement the integration** – Some combination of your engineers, perhaps including additional Guidewire Professional Services engineers.
- **Access to an experienced integration ISO consultant** – Due to the specialized nature of ISO integrations, each ISO integration team must have access to an experienced integration ISO consultant that they can contact for questions.

Requesting ISO testing and production server access

To properly test ISO integration, make a request to ISO to access the ISO ClaimSearch test server. ISO typically makes the test system ready to you within two business weeks of receiving a request. You must perform extensive testing with the ISO test server before moving any data to ISO final production with the production servers.

To integrate with production servers, you must make a request to ISO to access the ISO ClaimSearch production server, which is the public, actual database for your data. ISO typically makes the test system ready for you within two business weeks of receiving a request.

In your request to ISO to access to the ISO ClaimSearch test servers or production servers, you tell them your callback URL and an IP Address. The callback URL must be secured for HTTPS (SSL over HTTP) communication. An SSL certificate signed by a trusted authority must be used so that ISO can authenticate the URL. If a valid certificate is not already available, one must be purchased.

Whether or not a certificate is available yet, Guidewire recommends that you explicitly contact ISO to ensure that ISO recognizes the existing certificate or the about-to-be-purchased certificate.

Remember to have a valid agency ID string, which is stored in the ISO data path `MiscParty/ItemID/AgencyID`.

You must also tell ISO the acceptable IP address ranges for outgoing requests to ISO.

On the access form that ISO provides, there is an important property in the section “XML Request from Customer to ISO ClaimSearch.” There is a choice “Please indicate your preferred method of communication with ISO ClaimSearch.” For the choice “HTTPS Post,” select “No.” For the choice “Web Services using SOAP,” select “Yes.”

ISO configuration and integration testing

Carefully read the entire topic for ISO configuration and integration information.

After you are ready for testing, thoroughly test ISO integration by using the ISO test servers. Because of the complexity of ISO, this process sometimes takes longer than initially planned. Guidewire recommends that you budget time for extensive testing with potential delays as you refine your ISO configuration.

See also

- “ISO properties file” on page 603
- “ISO type code and coverage mapping” on page 607
- “ISO payload XML customization” on page 608
- “ISO match reports” on page 610

ISO integration data migration

If your legacy systems were integrated with ISO systems, you may need to update ISO data. Specifically, you must mirror changes that you make to claim data as you convert from a pre-ClaimCenter system to Guidewire ClaimCenter by updating ISO database records.

Along with ISO data format differences, any changes in ISO feed types affect the complexity of the data migration.

See also

- “ISO formats and feeds” on page 613

ISO production testing, including connectivity testing

Before the final transition to the production server, Guidewire strongly recommends a connectivity test to confirm the network configuration, including firewalls and proxies. At the minimum, be sure to do a minimal test with the validation URL with “Message 1: Claims Sent.”

Test connectivity with the validation URL before submitting claims to the production database. Be careful not to submit any claims to the production server by using the wrong URL.

Additionally, you can submit claims to the production database to test validity and ensure they are rejected by purposely changing the address or city to be empty. ISO always rejects claims with an empty address or empty city.

If you want to temporarily cause this type of rejection, you can remove these properties by customizing the code that generates ISO payloads. ClaimCenter sends the claim to ISO and receives a reply to “Message 1: Claims Sent.” Next it sends an additional “Message 2: Claims Processed” message and Message 2 contains a rejection.

If you do not ever receive Message 2, your system has at least one connectivity problem to resolve before going live with your production system. If you temporarily modify the payload to generate a rejection as discussed earlier, be extremely careful. Ensure that you undo any temporary changes before going live.

Run thorough tests and submit a few records to the production database before final deployment. Final deployment requires you to set the ISO URL to the submission URL, not the validation URL.

Coordinate ISO production server tests carefully with ISO to reduce the chance of accidentally corrupting ISO's production database with invalid data.

To change the ISO URL that ClaimCenter uses, change the `ISO.ConnectionURL` setting within `ISO.properties`. You may need to change firewall and proxy settings in conjunction with this URL change.

See also

- “ISO network architecture” on page 587

ISO production deployment

After completing production testing, your systems are ready for production deployment.

ISO network architecture

ClaimCenter includes built-in support for communicating with ISO. However, ISO's asynchronous responses require special network configuration including a network proxy to safely and efficiently handle ISO responses.

There are two main issues involved.

1. **For security, a network proxy insulates internal networks from external messages** – ISO must send its responses to a your callback URL. You must expose the callback URL outside the your Internet firewall. If you do not want to expose your ClaimCenter URL outside the firewall, you must have another server acting as the proxy.
2. **For performance, off-load SSL to a server other than ClaimCenter** – Although ClaimCenter can use SSL for outgoing requests, it is most efficient to handle SSL encryption outside ClaimCenter because Java-based SSL uses large processing resources. Instead, encode and decode SSL by using a separate proxy that can implement SSL faster, including native Apache web server support or other proxy systems designed for this task.

Both issues are solved by using an extra proxy server, sometimes called a bastion host, which lives in the area of the your network called the DMZ (De-Militarized Zone). The DMZ contains computers that are accessible from the outside world but partitioned off from your main network for network security. Network firewalls control access between the outside world and the DMZ, and also between the DMZ and the main network. The proxy server's job is to provide a secure gateway between an external service (in this case, ISO) and an internal server (in this case, ClaimCenter).

For outgoing messages from ClaimCenter, the proxy server forwards the request to ISO, and also typically wraps the request in an encrypted SSL/HTTPS connection. For incoming messages from ISO, the proxy server decodes the encrypted SSL/HTTPS request from ISO and forwards the request to the `ISOReceive` servlet, which is part of ClaimCenter.

Basic ISO message types

There are three basic messages between ISO and ClaimCenter.

Message 1: Claims sent

After you create a new claim, ClaimCenter sends the claim to ISO for processing in what this documentation refers to as “Message 1: Claims Sent.” Also, certain types of claim changes cause ClaimCenter to resend the claim to ISO. In this message, ClaimCenter sends a SOAP request over HTTPS to ISO with an XML payload that contains information about a new claim or changes to a existing claim.

The reply to this message (HTTPS request result) contains an ISO receipt with the following information.

- Indicates that data was received
- Validates the XML as well formed
- Confirms a valid ISO customer ID
- Confirms the ISO customer password
- Confirms that the request came from a valid IP address for this ISO customer

Additional confirmation information is included in “Message 2: Claims Processed.”

For Message 1, ISO supports HTTPS POST format or SOAP format. However, ClaimCenter only supports SOAP format for Message 1. Inform ISO of your requirement for SOAP format during initial ISO registration on the form that they provide on the line that requests your preferred method of communication. You must inform ISO of their requirement for the SOAP format during initial ISO registration on the form that they provide.

Message 2: Claims processed

After “Message 1: Sending Claims” completes, including its HTTPS reply, ISO starts to process the new claims or the claim changes. Some time later, ISO finishes processing the request and asynchronously notifies ClaimCenter that the ISO databases now contain the new claim information. ISO initiates an HTTPS POST message to your callback URL confirming the claim submission or if there are errors. Possible errors could include incorrectly defined or missing properties that prevented claim submission or claim changes. This documentation refers to this message as “Message 2: Claims Processed.”

For Message 2, ISO always uses the HTTPS POST protocol, not the SOAP protocol. The choice mentioned earlier about the ISO registration form does not affect the use of POST protocol.

Message 3: Matches detected

At some future time, ISO might detect claims from other companies that match those submitted by your ClaimCenter implementation. This match can occur soon after ClaimCenter send the claims information, or it can occur much later, or it might not occur at all. If the match happens, ISO initiates an HTTPS POST message to your callback URL with information about the matching claims. This documentation refers to this message as “Message 3: Matches Detected.”

For Message 3, ISO always uses the HTTPS POST protocol, not the SOAP protocol. The choice mentioned earlier about the ISO registration form does not affect this use of POST protocol.

ISO network layout and URLs

From the ISO perspective, there are two basic types of URLs: the URLs from you to ISO and the callback URLs from ISO back to you. Guidewire strongly recommends that you put a proxy server between your ClaimCenter implementation and the ISO servers. Consequently, most proxy server configurations include the following basic network areas.

Your intranet

This internal network is not directly accessible from the Internet, and is the site of your ClaimCenter implementation. However, your intranet is accessible from your DMZ. The DMZ insulates your sensitive systems from hackers and other intrusions.

Your DMZ

The part of your network that isolates your intranet from the potentially dangerous Internet. Your DMZ is protected on both sides by firewalls. One firewall tightly controls access to and from the unsecured Internet. Another firewall tightly controls access to and from the DMZ to your intranet.

The Internet

Presume that the Internet is unsecured and dangerous. All connections over the Internet happen with secure sockets layer (SSL), which provide encryption and identity confirmation in HTTPS connections (SSL over HTTP).

ISO network

The ISO network as viewed from the Internet. It is protected by a firewall and exposes only a handful of IP addresses to the outside world. One exposed IP address is the server that handles incoming requests. Another IP address (although theoretically it could be the same IP address) is the computer on ISO's network that performs callbacks to your ClaimCenter implementation.

With these network areas in mind, note the four different URLs in ISO communication. The following list describes each one. After the list is a diagram of the network architecture showing each URL. The URLs include placeholder variables by using a dollar sign (\$), which is the syntax used in the example Apache directive configuration files included with ClaimCenter.

The URLs to be configured are described below.

URL #1

The outgoing request for "Message 1: Claims Sent" as viewed from your intranet. The port number is configurable. The format of the URL is: `http://$DMZProxyDomainName:$DMZProxyPortA`. The proxy server translates this URL into URL#2. The ISO receipt in the response of URL#2 becomes the ISO receipt response for URL#1.

URL #2

The outgoing request for "Message 1: Claims Sent" defined by ISO. This URL, one of several ISO-specified URLs, always uses HTTPS and is always on port 443. Use different URLs depending on whether you are validating basic connections, testing submissions with the ISO testing server, or testing with the ISO production (real) server.

Testing URL

Use `https://clmsrchwebsvct.iso.com:443/ClaimSearchWebService/XmlWebService.asmx` if calling ISO for programmatic test submission. The XML message validates and the claim submits to the ISO test database. ISO returns "Message 2: Claims Processed" after the claim inserts into the database and one or more instances of "Message 3: Matches Detected" after matches are found. This URL always uses the test database, not the production ISO database.

Validation URL

Use `https://clmsrchwebsvc.iso.com:443/ClaimSearchWebService/TestUtility.htm` if calling ISO for programmatic validation. ISO tests that validity of your XML message and authentication without submitting the claim to any ISO database. Although there is an immediate reply to the request containing a receipt, ISO never sends return messages as a consequence of calling the validation URL. This URL technically uses the production database but never affects the database content since no claim submits to the database.

Submission URL

Use `https://clmsrchwebsvc.iso.com:443/ClaimSearchWebService/XmlWebService.asmx` if calling ISO for programmatic submission to the production (real) database. The XML message validates and the ClaimCenter submits the claim to the ISO production database. ISO asynchronously returns "Message 2: Claims Processed" after the claim inserts into the database. ISO might later send one or more of "Message 3: Matches Detected" after finding matches.

In all URL variants listed, the HTTP/HTTPS reply receipt indicates the validity check success. The validity check includes verifying well-formed XML syntax, ISO customer ID, ISO customer password, and the authorized IP address range. The complete ISO URL is referred to in the example Apache directive configuration files as `$ISO_URL`.

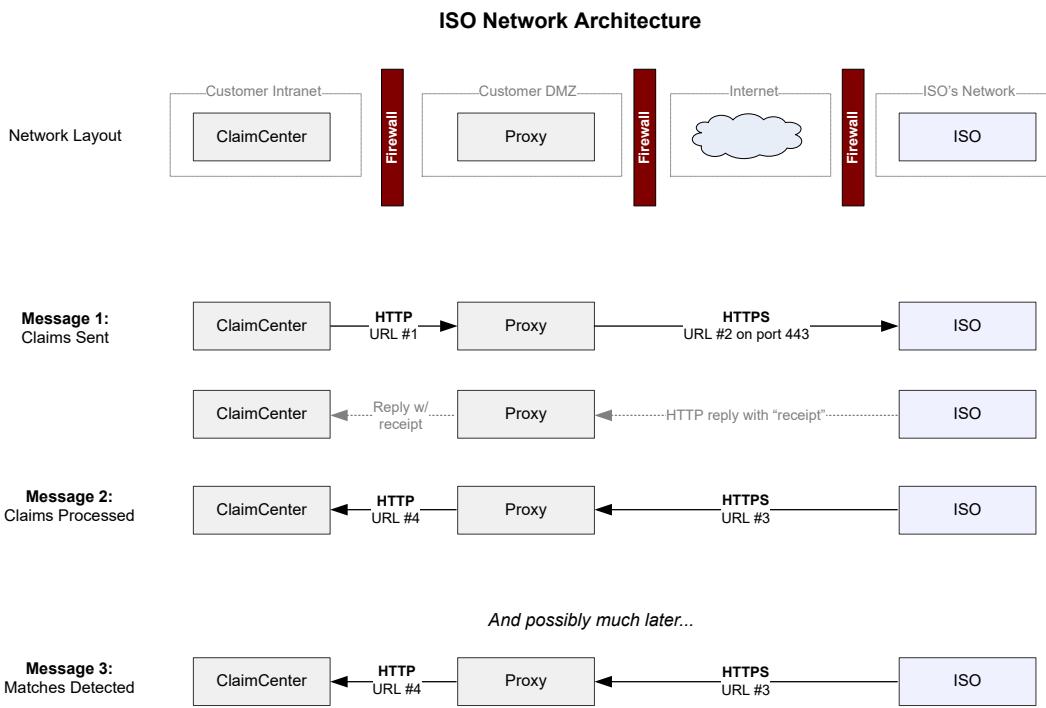
URL #3

The ISO callback URL for "Message 2: Claims Processed" as viewed from the public Internet. This URL typically points directly to your main external firewall. The port number is configurable. The format of the URL is `https://$DMZProxyDomainName:$DMZProxyPortB`. Ask ISO what IP address from which they send the callbacks. This IP address, which may not have a corresponding domain name, is referred to in example Apache directive configuration files as the IP address `$ISOCallbackIP`. The immediate HTTP reply to this request does not contain significant information.

URL #4

The ISO callback URL for “Message 2: Claims Processed” to the ClaimCenter server as viewed from your proxy server in the DMZ. The port number is configurable. The format of the URL is `http://server:port/cc/ISOResponse`. The immediate HTTP reply to this request does not contain significant information.

The typical ISO network setup, the three ISO messages, and the four ISO URLs are illustrated in the following diagram. Solid black arrows in the diagram show the primary direction for the HTTP/HTTPS request, but in all cases there is a synchronous HTTP/HTTPS reply. However, only for “Message 1: Claims Sent” is the content of the reply used at all.



ISO and ClaimCenter clusters

If you use ClaimCenter clusters, the only servers in the cluster that interact with ISO from a network architecture standpoint are servers with the **messaging server** role.

If a user makes a change on a claim or an exposure that triggers sending an exposure to ISO, that change submits a message to the messaging send queue. ClaimCenter stores the message in the database as a **Message** entity. The ISO messaging code that runs ISO-related rules runs on the server that triggered this change. However, after submitting the message to the send queue, ISO communication is managed only from the servers with the **messaging server** role.

There are two implications for your implementation and network architecture.

- **Outgoing messaging** – The messaging send queue and messaging plugins run only on servers with the **messaging server** role. Only those servers send outgoing messages to ISO through your proxy server.
- **Incoming messaging** – The ClaimCenter ISO servlet runs only on servers with the **messaging server** role, waiting for match requests from ISO through your proxy server. Ensure that your network proxy sends callback requests only to servers with the **messaging server** role.

You can additionally limit which cluster members support ISO callbacks by editing the Plugins registry in Studio. Navigate to **configuration > config > Plugins > registry** and open `IISOReplyPlugin.gwp`. Set the **Server** field to a specific server ID. Similarly, you can optionally use the **Environment** field. If the ISO servlet receives a reply from ISO but the message reply plugin is disabled on that cluster member, the servlet returns an error to ISO.

Configure your server roles, firewall settings, and proxy servers accordingly.

ISO activity and decision timeline

This topic describes the process of sending details of an exposure or claim to ISO and getting responses.

Initial ISO validation

ISO initial validation proceeds as follows:

1. The exposure is created.
2. The exposure is validated to level ISO.
 - a. The ISO validation rules run and verify that all properties needed by ISO are present and correct.
 - b. The ISO message rules examine the exposure and construct an appropriate message payload, which ClaimCenter puts on the central message send queue in the database.
3. On servers with the messaging server role, ClaimCenter gives the ISO messaging destination (the messaging plugins) one message to send from the message queue. The destination sets a few properties in the payload—for example, the message ID and the property indicating whether the message is an *add* (initial search) or *replace* operation. Because this claim or exposure is not marked as Known to ISO, the destination marks this message as an add operation.
4. The ISO destination sends the message to ISO by using SSL. It receives an immediate receipt indicating whether the message is correctly formatted and contains the correct authentication information. If the receipt is bad, the destination adds a non-retryable error acknowledgment to indicate that resending the message does not help and the configuration probably needs to be fixed.
5. The ISO destination waits for a response from ISO. The initial receipt is not an acknowledgment, although it can be treated as an error acknowledgment if it is bad. Only the full response is considered to be an acknowledgment.
6. The ISO server processes the request and sends a response to the bastion host SSL callback URL.
7. The bastion host forwards the response on to the `ISOReceiveServlet` running in the ClaimCenter server.
8. The `ISOReceiveServlet`:
 - a. Authenticates the response by checking the password, message ID, and other properties.
 - b. Parses the response.
 - c. Notifies the ISO destination.

For most of the actual reply handling, the servlet delegates the work to the `ISOReplyPlugin` class. The `ISOReplyPlugin` class is a `MessageReply` plugin implementation responsible for the asynchronous message reply.

9. The `ISOReplyPlugin` message reply plugin marks the original message as acknowledged.
 - If the response was good, the message marks the claim or exposure as Known to ISO, meaning that future requests are replace requests rather than add requests.
 - If the response contained errors, such as missing required properties, then a non-retryable error acknowledgment is submitted. Resending the message does not help and you are likely to need to correct the configuration.
10. If the response was good, the `ISOReplyPlugin` sets the ISO receive date on the claim or exposure, adds the match report document, and creates `ISOMatchReport` entities for any match reports.
11. During message reply handling in the `ISOReplyPlugin` plugin, ClaimCenter adds any activities necessary to review the claim or exposure.

Simple claim change

1. User changes a property on exposure or an associated claim, policy or claim contact, causing a change event.

2. ISO business rules check if any property important to ISO changes. If so, constructs a message payload and adds it to the message queue.
3. Similar to the “Initial Validation” steps listed earlier as steps 3 through 11. However, ClaimCenter marks it as a replacement because the claim or exposure property for “known to ISO” is set to `true`. If any match reports are returned, ClaimCenter adds another document, but only adds new `ISOMatchReport` entities if they differ from the previously added `ISOMatchReport` entities. If they do not differ, ClaimCenter updates the existing entity’s received date, but does not create a new entity.

Simple key field change

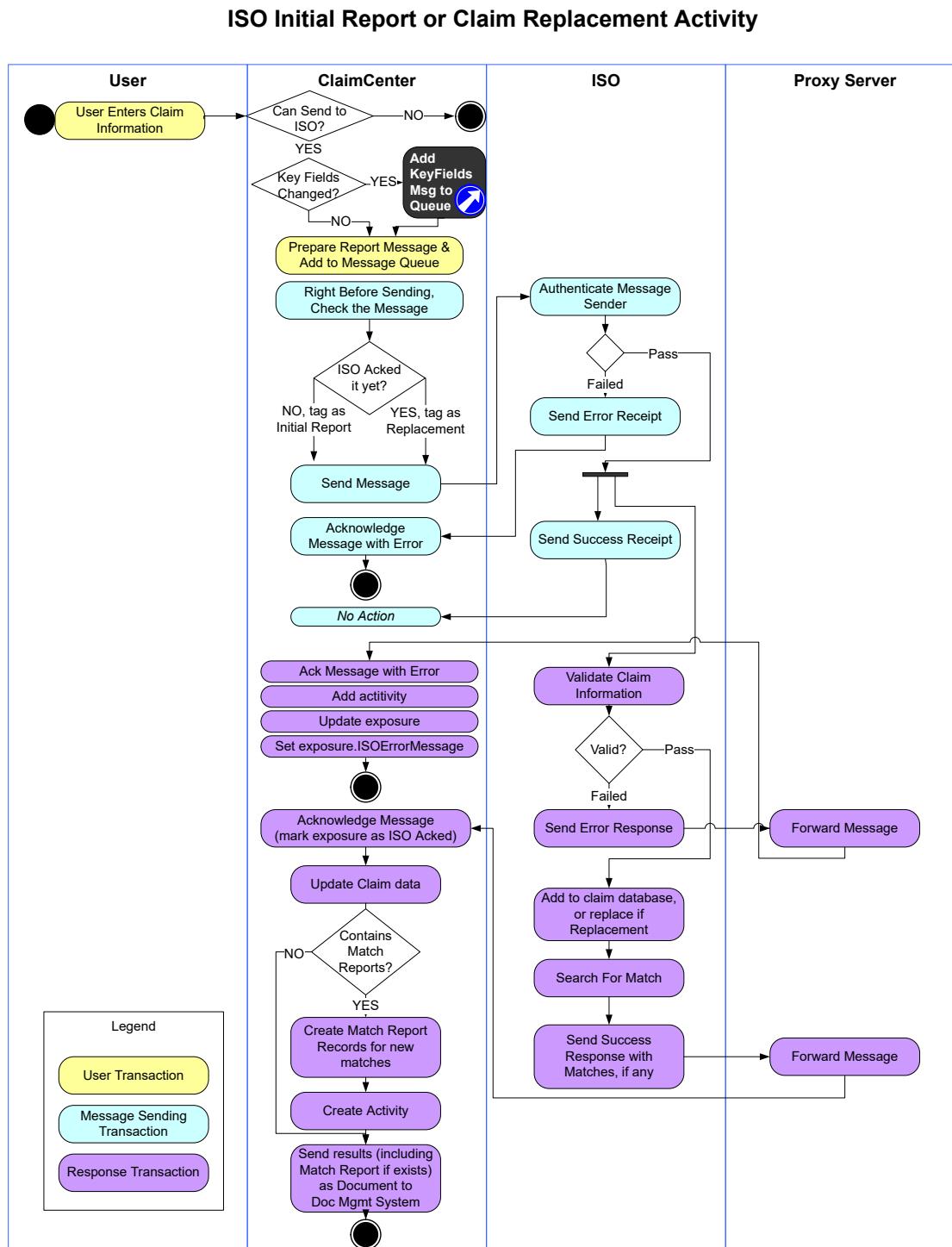
ISO uses a set of key fields to identify a claim. These key fields are the claim number, policy number, agency ID, and loss date (only the date matters, not the time of day). If the user changes any of these fields (loss date is the only property currently available through the user interface), the following sequence of events occurs.

1. User changes a key field.
2. ISO destination rules detect that key field has changed, constructs special “key field update” message payload and adds it to the message queue.
3. ISO destination sends the special payload to ISO, handling the ISO receipt normally.
4. If a key field update message succeeds, ISO does not send a response. The ISO destination sets a timeout configurable by using the `ISO.KeyFieldUpdateTimeout` property. If no error response is received within the timeout, the destination assumes the message has succeeded.

After the key field update message is handled another “replace” message is sent to get any new match results.

ISO activity and decision diagrams

The following diagram describes the ISO initial report and claim replacement activity. They are basically the same flow. The main difference is that immediately before the next message is sent from the message queue, the ISO message-sending code checks if that exposure was acknowledged by ISO. If it was, it marks a flag in the ISO payload as a claim replacement instead of a new claim.

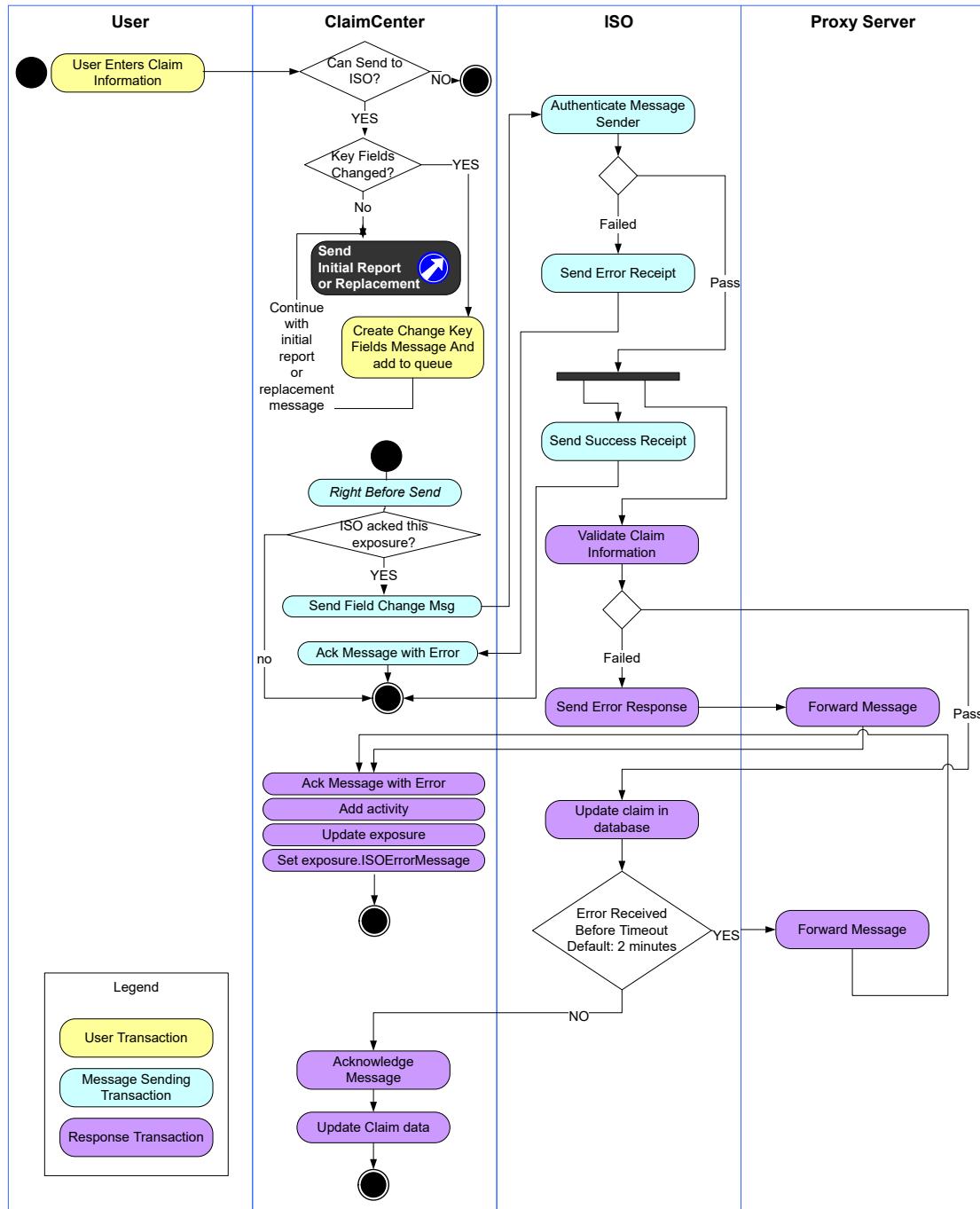


The following diagram describes the activity during a key fields change message. A key fields change message is a special type of message that ClaimCenter sends to ISO. For claim-based ISO messaging, ClaimCenter sends a key fields message after any key fields change on a claim. For exposure-based ISO messaging, ClaimCenter sends a key fields message after any key fields change on an exposure.

Because ClaimCenter and ISO use these properties to track identity and uniqueness for the entities in ClaimCenter, ClaimCenter must tell ISO about the change. If ISO has not heard of the item yet, any changed properties are irrelevant.

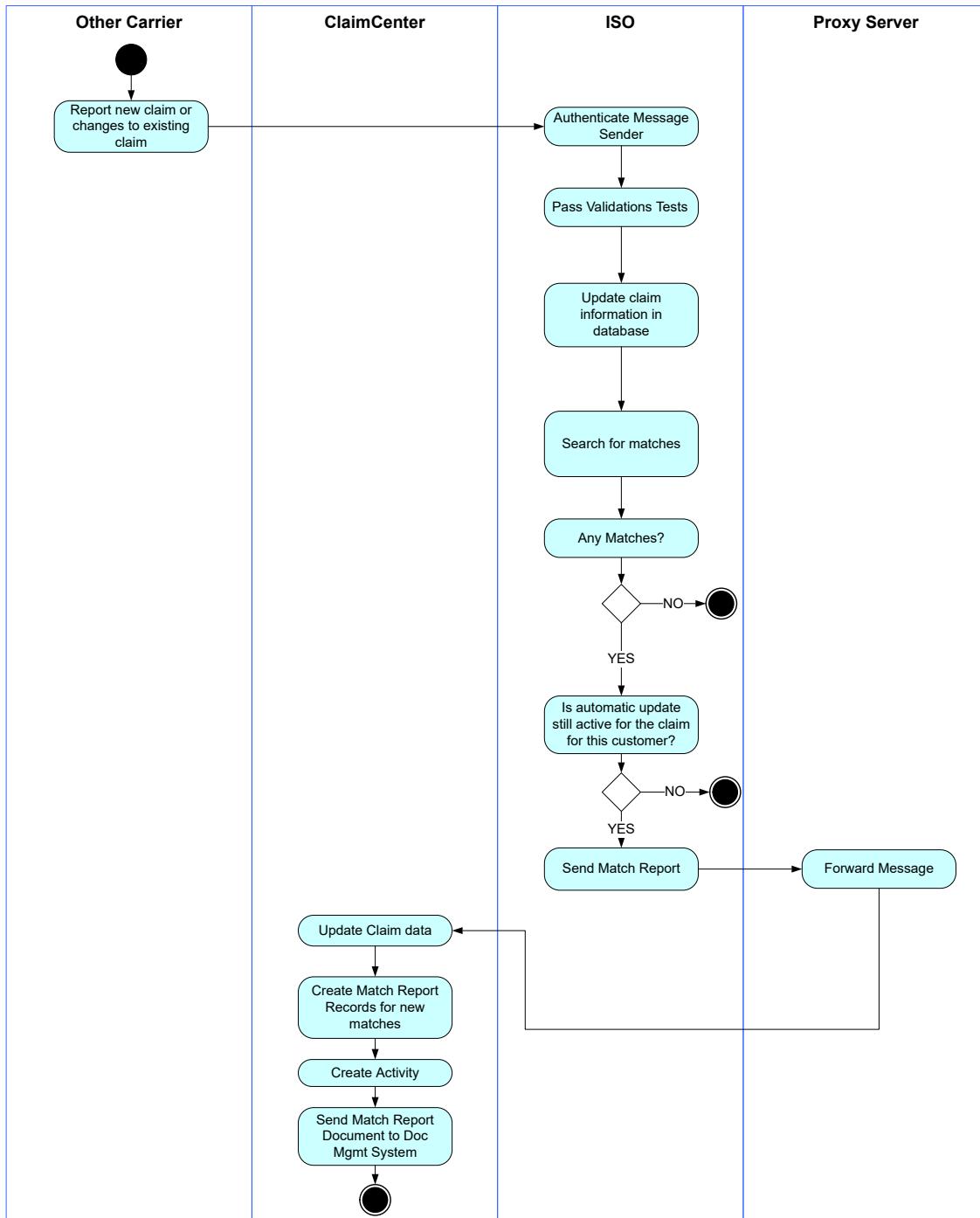
to ISO. Immediately before the message is sent, ClaimCenter checks whether ISO acknowledged that exposure by checking the KnownToISO property on the claim or exposure. If ClaimCenter thinks ISO knows about the claim or exposure, then it sends the key fields change message. Otherwise, ClaimCenter ignores the key fields message and does not send the message to ISO.

ISO Key Fields Changed Activity



The following diagram describes the activity during a post-submit scan for matches after another insurer finds a match.

ISO Post-Submit Match Report Activity



See also

- The diagram with Message 3 in the section “ISO network architecture” on page 587

ISO authentication and security

All ISO communication across the unsecured Internet uses secure sockets layer (SSL) connections to protect claim data. SSL lets the message sender and receiver confirm the other's identity and guarantees that no one can eavesdrop on the connection. For example, the ISO destination can be sure it is communicating with ISO because only ISO has the private encryption key associated with the certificate of <https://clmsrchwebsvc.iso.com>. Similarly, ISO can ensure it is connecting to the correct callback URL because only you have the private key associated with the certificate of the callback URL. Both ISO and you have your own public/private encryption key pairs and associated certificates signed by trusted parties.

The proxy server (not ClaimCenter) is the server that must contain all the appropriate encryption keys and certificates to ensure safe and secure SSL connections. Therefore, the proxy server is the main component of the ISO architecture that must be hardened to avoid unauthorized access to the sensitive private keys.

In the setup described earlier, connections between ClaimCenter and the proxy server are not encrypted with SSL. Not using SSL is the typical setup because intranet connections typically are presumed to be secure. Also, this approach reduces server processing load for encrypting SSL/HTTPS data on the ClaimCenter server. Performing SSL encryption within the Java virtual machine is particularly resource intensive. If you have some special reason for encrypting data between ClaimCenter and the proxy server, it is possible. However, SSL configuration is more complex and this approach is not fully discussed in the *Integration Guide*.

ISO security with customer IDs and IP ranges

All requests to ISO must contain a valid ISO customer ID. Because ISO tracks a range of IP addresses for each ISO customer ID, ISO strictly requires requests contain a valid customer ID and comes from your appropriate IP address range. Because requests from ClaimCenter to ISO go through the firewall, ClaimCenter requests appear to come from the IP address of your firewall. As you register with ISO, you tell ISO an IP address range of IP addresses that are valid for proxy server outgoing requests.

Similarly, set up the proxy server so that it rejects responses that do not come from ISO's IP address. Confirm with ISO precisely what IP addresses are sources of the callback requests. These IP addresses do not need to be associated with externally-identifiable domain names (DNS names). You can hard code these IP addresses by number in your firewall configuration.

For the ISO production servers, confirm with ISO what IP address from which they send the ISO callbacks for "Message 2" and "Message 3." As of the publication of this documentation, the production IP addresses were 206.208.171.134, 206.208.171.63, and 206.208.171.89. These ISO IP addresses may not have corresponding domain names.

For the ISO testing servers (not the production servers), several different servers can send the callback messages. The IP addresses of the test servers are 206.208.170.244, 206.208.170.250, and 206.208.170.249. These ISO IP addresses may not have corresponding domain names.

Configure any of your firewalls, intermediate computers, or reverse proxies to allow incoming ISO messages from all of these servers.

If you have any problems with this type of configuration, or you think messages are coming in from other IP addresses, immediately contact ISO to confirm any changes in configuration. Do not allow incoming messages from other IP addresses that you might see, as they may be unauthorized requests from unauthorized IP addresses.

ISO security with customer passwords

All requests to ISO must contain a password. ISO checks that the password matches the customer ID and rejects the request if it does not match.

Similarly, the `ISOREceive` servlet, which runs as part of ClaimCenter to receive the ISO callback, rejects responses that do not contain the correct customer password. Therefore it is vital for you to protect your password in a configuration file on your ClaimCenter server.

After you receive your initial username and password from ISO (typically by using email), immediately change the password to avoid serious problems later on due to expiration of ISO-generated passwords. Once you change the password, then the ISO password never expires.

Change your ISO passwords by using two different methods.

- Tell ISO what passwords to use. Contact ISO directly by using phone or another secure system, not unsecured email.
- After you first log on to the ISO web site, specify a new password.

If you do not change the initial password immediately, then you must change it before it expires. Changing it immediately is better than waiting since the change itself can disrupt ISO service during the change. After you change the password, you might not receive some responses to some messages because the ClaimCenter password is out of sync with the initial password in the message.

ClaimCenter checks ISO responses to see if they contain the valid password. Because the message contains initial password instead of the new password, ClaimCenter rejects the responses. There is no easy workaround for this type of issue. Therefore, change your ISO password immediately to avoid more serious problems later on.

After you receive your initial password from ISO, immediately change the password to avoid serious problems related to password expiry.

Once changed, the password does not expire. Do not change the password again.

As you request access to testing servers or production servers, the request form that ISO requires includes the following question.

Do you wish to have ISO transmit an ID/Password/Domain back to your system for security?

This optional setting specifies whether to include additional information in the content of ISO-initiated messages.

However, these three items are unrelated to your standard ISO account ID or account password. All three fields in the context of this question (ID, password, and domain) are arbitrary text fields that ISO offers to send with responses for extra authentication. The ClaimCenter ISO receive servlet checks those fields on incoming messages for authentication.

Answer “Yes” to that question on the form. Supply the desired password to ISO over the phone, which is their preferred approach for this security information. Guidewire recommends that you prepare for the possibility of eventual support for this feature by supplying text for these items during initial account setup. Remember that these can be three arbitrary text data. They do not need to match any other ISO account information.

See also

- Diagram with “Message 2” and “Message 3” in “ISO network architecture” on page 587

Optional support for HTTP basic authentication for outgoing requests

Separate from the main ISO user name and password settings in `ISO.properties`, you can optionally enable HTTP Basic authentication for outgoing requests.

Set the following `ISO.properties` properties.

- `ISO.Http.Authentication.Basic.Username` – user name
- `ISO.Http.Authentication.Basic.Password` – password

If both of these are non-null, ClaimCenter adds appropriate HTTP headers for HTTP Basic authentication.

This username and password combination can optionally be different from the `ISO.CustLoginId` and `ISO.CustPswd` properties.

ISO proxy server setup

The *proxy server* (also called a *bastion host*) forwards ISO responses to the server that is implementing (or testing) ClaimCenter ISO integration. As described earlier, the proxy server also is usually an intermediary computer for outgoing messages from you to ISO. In both cases, the proxy server hosts a *forwarder application*. This application takes incoming and outgoing requests, writes logs, and forwards requests to a port on another computer. It might also handle SSL encryption and decryption. Several common applications handle this type of forwarding, including the following applications.

- **Apache HTTP server** – Open source web server that can be configured as a proxy
- **Squid** – Open source dedicated proxy server
- **BorderManager** – Commercial product from Novell

From a technical standpoint, you can set up different proxy servers or applications to process different ISO messages. Do not set up proxy servers in this way unless you have a special requirement to do so.

For information about obtaining or setting up forwarding with these products, refer to the web sites and product documentation for those applications.

A common choice for a proxy server is the Apache HTTP Server. To simplify proxy server setup, this documentation includes “building blocks” of text to insert into Apache configuration file to facilitate proxy server setup.

See also

- “Proxy servers” on page 37

ISO validation level

The ISO validation level is used to confirm that a claim or exposure has all necessary information to submit it to ISO. ClaimCenter ensures that once a claim or exposure reaches this level, the validation level never drops below it at a later time. If the ClaimCenter user could remove properties needed by ISO or set to invalid values, it complicates the ClaimCenter ISO destination message plugin logic.

For claim-level messaging, a notable requirement in validation is that the claim description must be non-empty.

For exposure-level messaging, there is a notable requirement feature of the ISO exposure validation rules. The exposure validation rules strictly require the claim to already be at the claim validation level called ISO. The built-in rules already enforce this validation check. If you customize any of this code, you must continue to enforce this rule.

Validation timeline

After an exposure or a claim changes, ClaimCenter calls the validation rule set. In **Project** window of Guidewire Studio, navigate to **configuration > config > Rule Sets > Validation**. Open the file `ClaimValidationRules` to review the base ISO validation rules for claims. Similarly, navigate to **configuration > config > Rule Sets > Validation**, and open the file `ExposureValidationRules` to review the base ISO rules for exposures.

The validation rules may detect validation issues and may change the validation level for the claim or for the exposure.

After an exposure reaches the ISO validation level, ClaimCenter fires the Event Fired rule sets that detect this validation level change. These Event Fired rules send appropriate exposures to ISO. In practice, customizing these payload-generation rules for your data model and business requirements are the largest part of real-world ISO integrations. ClaimCenter provides payload generating Gosu rules for the most common payload types, and sends all required properties for those payloads. If you want to generate different ISO payloads, or add ISO optional properties, or add properties that ISO later makes required, customize the payload generation functions. In practice, the built-in rules provide nearly all of what most implementations need for initial deployment with the reference implementation data model and PCF files. Modify the rules to match your data model changes, generate new payloads, or add optional properties.

Checking priority of validation levels

In general if comparing typecode values, check the code of the typekey. However, validation levels are implicitly covered in a sequence. For example, reaching one level means that you have reached all levels before it.

Because it is important to compare relative position in the sequence, always check the `priority` property of a typecode to determine the validation level and compare to another value. Do not just check the equality of the `code` property of the typecode object because inequality does not indicate which level is higher.

For example, the exposure validation rules must confirm that the claim validation level is greater or equal the ISO level. Use the priority of the typecode, which is an integer that determines the ordering of the typecodes in that typelist. Check the `claim.ValidationLevel.Priority` value and see if it is greater than or equal to as `ValidationLevel.TC_ISO.Priority`. If it is, the claim is at least at the ISO validation level, or higher.

What are required properties for ISO?

The full ISO XML data hierarchy is described in the ISO-provided documentation called the “ISO XML User Manual.” The file name is `XML User Manual.doc`. ISO strictly enforces this hierarchy. Always request the latest version of this manual from ISO and refer to it during any customizations.

Adding additional properties to the payload, such as ISO optional properties not generated by the built-in rules, might affect your validation rules. For example, a precondition to sending a property to ISO is that it not be null. You add a validation to check if the property is non-null in validation rules prior to sending the property to ISO.

Validation level changes

You can add items to the validation level typelist or even switch values of validation levels by changing typecode `priority` properties in the typelist data model. For example, you could reverse the order of `iso` and `external` validation levels if necessary. If you make such changes, do so carefully and consider all possible other changes that you might need in your integration code or rules.

Make validation level or rules changes before deployment

Customize validation rules by changing the rules code or change validation levels by changing typecode `priority` properties in the typelist data model. As mentioned earlier, if you make such changes, do so very carefully and consider all possible other changes that you might need in your integration code or rules.

Be aware that making changes after deploying a production server is difficult. Make changes only after carefully considering the implications. For example, if you add a new validation requirement, some claims or exposures might not pass validation rules for the validation level as newly defined. You then get commit failures if anyone (a real user, batch process, or web service request) makes minor changes to an object in the validation graph. For example, validation rule changes could cause activity batch processes to fail on activities by adding a new claim validation rule that causes even trivial changes to a related claim.

Carefully design your validation rules and validation level changes before initial deployment. You can change them later but it may require major engineering and testing to ensure that claims and exposures never fail their current validation level after your changes.

ISO messaging destination

After you enable an ISO destination, the ISO destination requests notification for certain ClaimCenter events. Events are an abstraction of changes to Guidewire business data. The ISO destination listens for the messaging events `ClaimAdded`, `ClaimChanged`, `ExposureChanged`, and others. Event Fired rules create ISO-related messages as appropriate.

The ISO destination sends three subtypes of outgoing messages of type “Message 1: Sending Claims.”

Initial search request

Sent if the exposure becomes valid.

Replacement search requests

Sent if the exposure changes. These messages are nearly identical to the initial search request. A flag in the message indicates whether the information in this request is a new record or if it replaces previous information for the exposure.

Key field updates

Sent after changes to a key field, such as the claim number, policy number, agency ID, or loss date. These update messages are handled specially by the destination because ISO uses a different protocol for changes to properties that uniquely identify an exposure. A key field update is immediately followed by a replacement search request because key field changes might result in new matches.

The rules use a lot of shared code in a Gosu ISO library (the `libraries.ISO` class). It is likely that you will need to customize this code for your data model, particularly the `createPayload` method. The `createPayload` method creates the payload for a search request. The details of the payload depend on the your exposure types, policy types, coverage types, and coverage subtypes, so they could not be hard coded in the destination.

After the messaging destination sends a message to ISO, ClaimCenter gets an immediate receipt letting it know that ISO queued the request. If the receipt indicates a failure, then the destination adds an error acknowledgment immediately. Otherwise it waits for the following response, which is either a success (possibly containing match reports) or an error response. If the ISO destination gets a successful response, it adds an acknowledgment. If it gets a failure, it adds an error acknowledgment.

Key field updates are more complicated because ISO does not send a response if a key field update is successful. Instead, the destination waits for 2 minutes (a configurable delay), and if no error response appears within that time, the ISO destination submits the acknowledgment.

The destination tracks whether a particular claim or exposure ever had a successful initial search request and was successfully added to the ISO database. After it gets a successful response from ISO, the ISO destination sets the claim or exposure field `ISOKnown` to `true` and marks all future search requests as replacement search requests.

There are several possible error cases:

ClaimCenter cannot contact ISO

The inability for ClaimCenter to contact ISO for the send request is considered a temporary retryable error. The destination uses the standard destination backoff delay and retries with increasing delay time.

ISO sends a failure receipt

Failure receipts indicate a non-retryable error. Something must be wrong with the configuration for ISO to reject the request. The ISO destination submits an error (negative) acknowledgment for the message, and then an administrator must diagnose what went wrong, fix the configuration problem, and send a new ISO message.

If the problem is with the message payload due to incorrect design of business rules, a common problem during development, do not retry the message. Instead, delete the message and send it again by modifying the data or by using the **Send to ISO** button in the ClaimCenter user interface.

ISO sends a failure response

Failure responses indicate a non-retryable error. See the previous failure receipt description.

ClaimCenter is down, preventing ISO from responding

If ISO cannot reach ClaimCenter, ISO usually retries for at least 12 hours. This retry period is typically an adequate length of time to wait for ClaimCenter to be back up.

The ISO destination and its associated classes have their own logger category. Setting `log4j.category.Integration.messaging.ISO` to `DEBUG` generates log messages whenever an ISO message is sent or received. There is also the ISO property `ISO.LogMessagesDir`, which causes all ISO messages and responses to be logged to a directory.

If you use ClaimCenter clusters, data changes can happen on any server in the cluster. Consequently, any server can trigger ISO rules that generate outgoing messages to ISO. Although any server can add a message to the messaging send queue, the actual outgoing communication between ClaimCenter and ISO happens only on servers with the

messaging server role. The messaging plugins for outgoing ISO messages and the ISO receive servlet for ISO responses run only on servers with the messaging server role.

ISO receive servlet and the ISO reply plugin

The ISO receive servlet on servers with the messaging server role handles all asynchronous responses from ISO to ClaimCenter. ISO sends these responses to the bastion host, and the bastion host forwards responses to the receive servlet.

The servlet parses each response and then performs the following actions.

1. The servlet checks that the response contains a valid request ID and the correct password. If not, the response is rejected and an error is written to the log.
2. The servlet notifies the ISO destination, in case it is waiting for an acknowledgment.
3. The servlet updates the `ISOReceiveDate` field on the claim or exposure.
4. The servlet adds the response XML as a document on the claim or exposure.
5. If the response contains any match reports, the servlet adds `ISOMatchReport` entities to the `ISOMatchReports` array on the exposure. The servlet checks to see if each `ISOMatchReport` is the same as an existing one. If it is the same as a previous one, ClaimCenter updates the existing entity's `ReceivedDate` instead of creating a new entity.

For example, suppose a claim or exposure changes several times and always gets the same two match reports back after each change. Afterward, the claim or exposure contains only two `ISOMatchReport` entities.

6. If ClaimCenter receives any match reports, the receive servlet calls `ISOReplyPlugin` to add activities.

If you use ClaimCenter clusters, data changes can happen on any server in the cluster. Consequently, any server can trigger ISO rules that generate outgoing messages to ISO. Although any server can add a message to the messaging send queue, the actual outgoing communication between ClaimCenter and ISO happens only on servers with the messaging server role. Messaging plugins for outgoing ISO messages and the ISO receive servlet for ISO responses run only on servers with the messaging server role.

However, for most of the actual reply handling, the servlet delegates the work to the `ISOReplyPlugin` class. The `ISOReplyPlugin` class is a `MessageReply` plugin implementation responsible for the asynchronous message reply.

See also

- “ISO network architecture” on page 587

ISO properties on entities

ClaimCenter includes ISO-specific properties and methods on both claims and exposures.

For new claims, ClaimCenter always send claims not exposures to ISO. Exposure-based messaging is only supported for claims with previously-reported exposures by earlier versions of ClaimCenter.

From an implementation perspective, the similarities between exposures and claims are defined by the fact that `Claim` and `Exposure` entities both implement the interface `ISOReportable`. Be aware of this implementation as you review the documentation or you are looking through built-in code such as payload generation Gosu code. You can generally think of a reference to `ISOReportable` as a claim or exposure. Whether ClaimCenter actually uses claim-level messaging is controlled by the `ISO.properties` file property `ClaimLevelMessaging`. Additionally, for any legacy exposures that are known to ISO before switching to claim-based messaging, ClaimCenter continues to treat these as exposure-based messaging.

The following table lists ISO-related properties on exposures, claims, or vehicle entities. Note that some properties are exceptions and exist only on some entities and not others.

Entity	Property	Type	Description
Claim, Exposure	ISOCreateDate	date	The last time the claim (or exposure) was sent to ISO.
Claim, Exposure	ISOReceiveDate	Boolean	The last time a response was received by ISO for this claim (or exposure).
Claim, Exposure	ISOKnown	Boolean	A flag whether this claim (or exposure) was successfully received by ISO.
Claim, Exposure	ISOStatus	ISOStatus	The status of the claim (or exposure) with ISO. Choices are TC_None, TC_NotOfInterest, TC_ResendPending, and TC_Sent. The ISO status is “not of interest” (TC_NotOfInterest) if the ISO payload generation rules for an exposure return null, thus indicating that is not appropriate to send it to ISO. In some cases an exposure’s ISO status is “not of interest” but you might think that is incorrect. If so, check that it is a custom exposure type and the ISO payload rules were appropriately customized to generate any payload for that subtype. This property is viewable in the user interface for administrative users.
Claim, Exposure	ISOMatchReports	ISOMatchReport[]	An array of zero, one, or more ISO match reports for this exposure.

The following properties relate to ISO, but are not specific to ISO.

Entity	Property	Type	Description
Exposure only	LostPropertyType	LostPropertyType	For theft losses, the ISO category of lost property.
Vehicle	SerialNumber	varchar 2	The vehicle serial number. This is only used if the Vehicle Identification Number (VIN) is not appropriate, such as for boats.
Vehicle	BoatType	BoatType	If vehicle style is boat, this is the type of boat.
Vehicle	ManufacturerType	VehicleManufacturerType	The company that manufactured the vehicle, based on NCIC 2000 vehicle codes. This code is required by ISO, so ClaimCenter generates a similar (unofficial) code with the first characters of the Make property if the code is not present. If you implement ISO integration, you must ensure the code is set correctly.
Vehicle	OffRoadStyle	OffRoadVehicleStyle	The type of off-road vehicle, for instance snowmobile, All Terrain Vehicle (ATV), or other vehicle with wheels, or wheels and tracks. This is also based on NCIC codes.

ISO user interface

If you enable ISO integration, the biggest change is on the claim detail or exposure detail page. There is an **ISO** tab in the claim or exposure detail screen. This tab shows the ISO status, send and receive dates, and any match reports. You can select items on the match reports list view to see details of a match.

For new claims, always send claims not exposures to ISO. Exposure-based messaging is only supported for claims with previously-reported exposures by earlier versions of ClaimCenter.

Remember that whether ClaimCenter actually uses claim-level messaging is controlled by the `ISO.properties` file property `ClaimLevelMessaging`. Additionally, for any legacy exposures that are known to ISO before switching to claim-based messaging, ClaimCenter continues to treat these as exposure-based messaging.

Specific ISO views that are built-in to ClaimCenter and can be customized within the configuration module in the product directory `ClaimCenter/modules/configuration`.

Description	Location in configuration module
ISO detail view for claims or exposures	config/web/pcf/claim/exposures/iso/ISODetailsDV.pcf
ISO list view for displaying a list of match reports	config/web/pcf/claim/exposures/iso/ISOMatchReportsLV.pcf
ISO detail for displaying an individual match report	config/web/pcf/claim/exposures/iso/ISOMatchReportDV.pcf

These are standard Guidewire PCF configuration files, so you can configure them like other PCF files or remove them entirely if you do not need them. By default, all user interface to these properties are read-only. The user interface displays status information and ISO information. In generally, you do not need to change these files.

If you have the Integration Admin permission, you see an extra property (ISO known) and can view and edit the ISO known and ISO status properties (`ISOKnown` and `ISOStatus`). This information can be useful as you correct configuration errors. For example, log in as an administrator with the Integration Admin permission. If you can see if the exposure status is incorrect, for example if the status is stuck at “Not of interest to ISO”. The application displays the status “Not of interest to ISO” if the payload generation function returns `null`. Typically the functions return `null` by accident because you added a custom exposure type and you did not yet modify the built-in payload code to handle it.

If the ISO destination is enabled, in the exposure detail toolbar there is a **Send to ISO** button. The button is enabled after the exposure was validated and sent to ISO. Use it to manually resend the exposure to ISO. Most changes to the exposure/claim/policy also cause a resend to ISO. You only need the button if you want to send the exposure to ISO but you are not changing any properties.

ISO properties file

The `ISO.properties` file is found in `ClaimCenter/modules/configuration/config/iso/ISO.properties`. For debugging-only modifications, note that ClaimCenter reads this file at the time ClaimCenter initializes the destination but rereads it if the ISO destination suspends then later resumes.

The name of the ISO property file is configurable by using the `config.xml` file property `ISOPropertiesFileName`.

The following are the most critical `ISO.properties` settings for a typical ISO deployment.

- `ISO.AgencyId`
- `ISO.ConnectionURL`
- `ISO.CustLoginId`
- `ISO.CustPswd`
- `ISO.RequireSecureReceive`

The following table contains a reference for properties in the `ISO.properties` file.

Property	Description
<code>ISO.AgencyId</code>	ID assigned by ISO to identify the company and office submitting a request. This property is required and has no default value, so it must be specified in <code>ISO.properties</code> . For claim-level ISO messaging, you can instead specify this property at the claim level.
<code>ISO.ClaimExposureNumberSeparator</code>	The separator text used to separate the claim number from the exposure order number for building a unique identifier for an exposure. ClaimCenter sends this unique identifier to ISO. For example, if the separator is “exp”, the unique identifier for exposure 1 on claim 123-45-6789 is the value 123-45-6789exp1. ISO strips out all non-alphanumeric characters from the unique identifier, so the example resolves to 123456789exp1 after processing by ISO. Your separator text must be alphanumeric. This property applies only to exposure based ISO messaging. For claim-based messaging, ClaimCenter ignores this property.

Property	Description
ISO.ConnectionURL	<p>The URL of outgoing requests to ISO server. For the recommended server architecture, this URL is URL#1. This URL is the URL from ClaimCenter to the proxy server.</p> <p>This property defaults to the acceptance (test-only) service URL: <code>https://claimsearchgwa.iso.com/xmlsoap</code></p> <p>For production (non-test) use after testing is complete, set to the service URL: <code>https://claimsearchgwp.iso.com/xmlsoap</code></p> <p>If you use a proxy server between ClaimCenter and ISO, set this property to use your proxy server and its port number—<code>http://proxyDomainOrIP:portnumber</code>. Using the syntax of the Apache configuration file example file, use the URL <code>http://\$DMZProxyDomainName:\$DMZProxyPortA</code></p>
ISO.CurrencyCode	Your currency code, defaults to en_US. Not usually changed.
ISO.CustLangPref	Your language preference, defaults to en_US. Not usually changed.
ISO.CustLoginId	The login ID that is set by an implementation of the <code>CredentialsPlugin</code> . See “ Credentials management ” on page 605
ISO.CustomerPhoneFormat	<p>Regular expression used to parse phone numbers from your ClaimCenter implementation. ISO requires phone numbers in a special format: +1-650-3579100. The phone format is used to parse a ClaimCenter phone number, pull out the area code and remaining 7 digits, and then convert it to the ISO form. The regular expression must match three groups of numbers - the area code, then the remaining blocks of 3 and 4 digits. The default format is “([0-9]{3})-([0-9]{3})-([0-9]{4})(x[0-9]{0,4})?”. This format handles numbers of the form 650-357-9100 with an optional 0-4 digit extension.</p>
ISO.CustPswd	The password that is set by an implementation of the <code>CredentialsPlugin</code> . See “ Credentials management ” on page 605
ISO.EncryptionTypeCd	Encryption mode, defaults to NONE. Not usually changed.
ISO.ExpectReplies	<p>This flag defines whether the destination expects replies from ISO. Set this flag to true for production servers. If false, the destination sends messages to ISO in test mode, which generate no responses nor adds anything to their database. However, the ISO destination gets an immediate receipt from ISO confirming that the connection to ISO works and that the XML syntax and authentication information were correct.</p>
ISO.Http.Authentication.Basic.Password	<p>Password for HTTP Basic authentication for outgoing calls to ISO. If <code>ISO.Http.Authentication.Basic.Username</code> and <code>ISO.Http.Authentication.Basic.Password</code> are non-null, ClaimCenter adds appropriate HTTP headers for authentication. This username and password combination can optionally be different from the <code>ISO.CustLoginId</code> and <code>ISO.CustPswd</code> properties.</p>
ISO.Http.Authentication.Basic.Username	<p>Username for HTTP Basic authentication for outgoing calls to ISO. If <code>ISO.Http.Authentication.Basic.Username</code> and <code>ISO.Http.Authentication.Basic.Password</code> are non-null, ClaimCenter adds appropriate HTTP headers for authentication. This username and password combination can optionally be different from the <code>ISO.CustLoginId</code> and <code>ISO.CustPswd</code> properties.</p>
ISO.KeyFieldUpdateTimeout	Number of seconds to wait before assuming that a key field update request has succeeded so it can proceed with the next request. Defaults to 120.
ISO.LogMessagesDir	The name of a directory to log outgoing and incoming ISO XML messages. Defaults to null if unspecified. Messages are saved to files named after their message id, for example <code>1_request.xml</code> , <code>1_receipt.xml</code> and <code>1_response.xml</code> .
ISO.MatchReportNameFormat	Simple date format used to generate the name for the match report document. Default is “ <code>ISOMatchReport-'yyyy-MM-dd-HH-mm-ss'.xml</code> .”

Property	Description
ISO.Name	Client application name. Set to XML_TEST during testing and but also during production. This property is required and has no default value. You must specify this value in ISO.properties.
ISO.NameSpace	URL namespace for the ISO claim search SOAP service. Defaults to http://tempuri.org/ and usually does not need to change.
ISO.Org	Client application ISO Organization code, defaults to ISO. Not usually changed.
ISO.RequireSecureReceive	Defines whether the receive servlet require that all incoming requests (from the point of view of the ClaimCenter server) are on a secure (HTTPS) connection. For the recommended server architecture, set to false. In this approach, the ClaimCenter server itself does not take on the burden of SSL processing within the Java virtual machine. Instead, the proxy server provides the SSL/HTTPS processing. The default of this property is true.
ISO.SendMessages	Configures whether the destination sends messages to ISO. Set to true in production. If false, the ISO destination drops messages. Set this property to false only if debugging and logging all messages. Defaults to true.
ISO.SPName	Service provider name. Default is iso.com. This property is not usually changed.
ISO.TestSuffix	A number ClaimCenter appends to various properties on every request (the request id, claim number, policy number and insured's name) to make them unique in the ISO test database. Otherwise the results from one set of testing could interfere with the results from another set. Only for use during testing. During testing, start with this number at 1 and then increment it every time you drop a database and reimport data. If you do not, then ISO rejects all the sample data exposures because ISO thinks it has seen them already. Defaults to 0, which is the recommended number for production servers.
ISO.Version	Client application ISO version, defaults to 1.0. This property is not usually changed.

Credentials management

The ISO.CustLoginId and ISO.CustPswd credentials are set by an implementation of the `CredentialsPlugin`, the `SecretsManagerCredentialsPlugin.gs` class.

Each credential value (also known as a *secret*) is a combination of a username and password and follows the `login:password` syntax.

Note: You may also use an external substitution file in ClaimCenter to set credentials. See "External server configuration" in the *Administration Guide*.

IMPORTANT: Do not enter the ISO.CustLoginId and ISO.CustPswd credentials directly in the ISO.properties file as this method of storing credentials is not secure.

Populating match reports from ISO's response

A match report is information from ISO regarding other insurance companies with information about an exposure that seems similar to an exposure in your system.

You can customize how ISO match report properties in the incoming XML map to properties on the ISOMatchReport entity that stores the data on an exposure.

To do this, you can customize the ISOReplyPlugin class implementation in the `populateMatchReportFromXML` method. Perform any necessary logic there. For method arguments, this method gets the match report object to populate plus a Gosu object representing the match report XML. You can add as much additional post-reply logic you want.

See also

- “ISO match reports” on page 610

Converting from ClaimContact (ClaimCenter) to ClaimParty (ISO)

ClaimCenter exports exposure contact data to ISO. What Guidewire calls `ClaimContact` entities is approximately what ISO calls `ClaimParty` XML elements. Similarly, ClaimCenter maps what Guidewire calls `ClaimContact` roles to what ISO calls a claim party role code (`ClaimPartyRoleCd`). ISO considers this `ClaimParty` contact data optional but strongly recommended for effective claim matching, so ClaimCenter provides built-in support to generate these elements.

The most important part of configuring this conversion is defining mapping codes in your `ISO.properties` file. The default `ISO.properties` file has two contact-related sections to configure.

ClaimCenter uses these lines to map and generate new claim party XML elements.

```
ISOClaimParty.BS = repairshop
ISOClaimParty.CO =
ISOClaimParty.CT =
ISOClaimParty.FM =
ISOClaimParty.IB = agent
ISOClaimParty.LC = attorney
ISOClaimParty.LP = checkpayee
ISOClaimParty.MD = doctor
ISOClaimParty.MF = hospital
ISOClaimParty.PA =
ISOClaimParty.TW =
ISOClaimParty.OW =
ISOClaimParty.PT =
ISOClaimParty.TN =
ISOClaimParty.WT = witness
```

The code on the left is the ISO claim party code, and the code on the right of the equals sign is the ClaimCenter `ClaimContact` entity role typecode. Add or change these mappings as needed for any changed or added contact role typecodes. Each line can map to one or more ClaimCenter typecode. To include more than one, you can separate them with commas. If ClaimCenter matches a `ClaimContact` with this role typecode, ClaimCenter creates a new ISO `ClaimParty` element with that code.

However, the `ClaimContact` conversion is not a direct one-to-one conversion. There are several important things to know about this translation of `ClaimContact` data.

- ClaimCenter permits a `ClaimContact` to have multiple roles, as defined by an array of claim contact roles on a claim contact. ISO does not permit multiple roles for a `ClaimContact` in their data model. So, in some cases ClaimCenter creates multiple (ISO) `ClaimParty` XML elements.
- Currently, there is limited support for multiple contacts with the same role. In the current release, if more than one contact has a certain role (such as a witness or attorney), additional contacts with that role may be omitted from conversion.

Additionally, the following two lines configure special parts of the ISO payload designed specifically for witness and driver identification.

```
ISOClaimSearch.ContactRole.Witness = witness
ISOClaimSearch.ContactRole.Driver = driver
```

The code on the right of the equals sign is the ClaimCenter `ClaimContact` entity role typecode. For example, during ISO conversion, if a `ClaimContact` with the `driver` role typecode is found, it will be used as the ISO driver in the ISO `driver` XML element. The ISO driver corresponds to the `ClaimsDriverInfo` element in the `ClaimsParty` record. If a `ClaimContact` with the `witness` role typecode is found, ClaimCenter uses it for the `com.iso_AccidentWitnessedInd` element on the `ClaimsOccurrenceAggregate` element that describes this exposure/claim.

If more than one contact has a witness role or driver role, additional contacts with that role may be omitted from conversion. Also, as with the other `ClaimParty` mapping lines, each line can only map to a single ClaimCenter typecode.

ISO type code and coverage mapping

The ISO integration code includes an XML file to map ClaimCenter typecodes to typecodes that ISO understands. The type code mapping file is stored in `config/iso/TypeCodeMap.xml`. It uses the same format as the `typecodemapping.xml` file that is stored in the general-purpose file `config/ttypelists/mapping`. However, this file is used only by the ISO destination so its mappings must have their namespace attribute set to `iso`.

The type code mapping file is read by the ISO messaging destination as it starts. For debugging use only, be aware ClaimCenter rereads this file if the destination suspends and later resumes.

The important type codes are listed in the following table. Note that `PolicyType`, `CoverageType`, and `CoverageSubType` are not listed because those use a separate file, discussed later in this topic.

ClaimCenter typecode Maps to ISO typecode	
LossType	ISO Loss Type code
ExposureType	ISO Subject Insurance code. This mapping indicates whether the damage is to property, contents, or loss of use. Used only for property exposures.

It is possible to add other typecodes to the file, but in practice it typically is not necessary.

Coverage mapping

The ISO Policy Type → Coverage → Loss Type hierarchy is very similar in concept to the ClaimCenter coverage subtype hierarchy (`PolicyType` → `CoverageType` → `CoverageSubType`). The full ISO XML data hierarchy is described in the ISO-provided documentation called the “ISO XML User Manual.” The filename is `XML User Manual.doc`. ISO strictly enforces this hierarchy. Always request the latest version of this manual from ISO and refer to it during any customizations.

ISO rejects any requests if the loss type is not in the list of permitted loss types for the given coverage. This requirement can make it challenging to map the ClaimCenter hierarchy to the ISO hierarchy, depending on how closely your data model hierarchy matches ISO’s hierarchy. It is unnecessary for the map to cover all possible cases. You can override the ISO policy, coverage and loss type properties from business rules defined in Guidewire Studio.

The ISO integration uses a separate text file from the typecode mapping file to configure how ClaimCenter coverages map to ISO coverages. A comma-separated values (CSV) file called `ISOCoverageCodeMap.csv` controls this mapping.

ClaimCenter “coverage codes” represent the policy type, coverage type and coverage subtype. These map to the ISO policy type, coverage type and loss type.

The `ISOCoverageCodeMap.csv` file maps a ClaimCenter policy type, LOB code (optional), coverage type and coverage subtype to an ISO policy type, coverage type and loss type. The format is a comma-delimited row containing the following fields in the listed order.

- Source ClaimCenter policy type
- Source optional line of business code (can be blank)
- Source coverage type
- Source coverage subtype
- ISO policy type
- ISO coverage type
- ISO loss type

For example a couple of lines in this file might be similar to the statements shown below.

```
auto_comm,,ABI,abi_bid,CAPP,BODI,BODI  
businessowners,pr,ADPERINJ,adpersinj_gd,CPBO,OTPR,OTPR
```

Notice that the first example line does not include a line of business. The second line includes a line of business.

This format allows for hierarchy differences between the ClaimCenter and ISO policy type hierarchies, which were not supported by `TypeCodeMap.xml`. For example, ClaimCenter might share the same coverage type, `X`, between two different policy types, A and B. But ISO might have two different coverage types `ISOAX` and `ISOBX` to handle this case.

In the new coverage mapping system, you can now do the following mapping.

```
A,,X,CS,ISOA,ISOAX,ISOC
B,,X,CS,ISOB,ISOBX,ISOC
```

This mapping maps coverage type X to ISOAX if the policy type is A, but maps ISOBX if the policy type is B.

There are some special features to the `ISOCoverageCodeMap.csv` mapping.

- The ISO coverage code is empty in several mappings. These are usually mappings for first party property claims and exposures; ISO does not use a coverage code for such losses.
- The `LOBCode`. Normally ClaimCenter policy types map straight to ISO policy types, and entries for these policy types omit the `LOBCode` field. But there are some cases in which a single ClaimCenter policy type maps to multiple ISO policy types. In such cases the `LOBCode` can be added to the mapping, to narrow it down to a particular ISO policy type. For example, in the out of box configuration `businessowners,pr` (`businessowners` policy type, property LOB) maps to ISO's CPBO, while `businessowners,g1` maps to ISO's CLBO.
- In some cases the ClaimCenter coverage subtype is not very specific and the claim's loss cause gives a much better sense of the ISO loss type to use. The mapping file allows entries like `LossCause/OTPR` in the ISO loss type column. This entry tells ClaimCenter to try mapping the claim's loss cause field to an ISO value by using the `TypeCodeMap.xml` file. If no mapping is found default to the ISO loss type `OTPR`. This extended mapping is mainly used for homeowner's at the moment.

The ISO code uses a higher-level lookup operation based on a ClaimCenter policy type, LOB code, coverage type, coverage subtype, and loss cause by using a two-step full mapping lookup.

1. ClaimCenter looks for appropriate mapping lines in `ISOCoverageCodeMap.csv`.
2. If that lookup returns a loss type of the form `LossCause/OTPR`, ClaimCenter looks up the loss cause in the `TypeCodeMap.xml` mapping.

You can request a full lookup from Gosu by using the `ISOTranslate.getCoverageCodes()` method.

ISO payload XML customization

The largest part of real-world ISO integrations are customizing ISO payload-generation rules for changes to your data model and special business requirements, such as custom exposure types. Ensure you allocate adequate time for this process.

Within Guidewire Studio business rule sets, the ISO rules perform the following steps.

1. Validate a claim or exposure (and associated policy) to see if it is valid for ISO.
2. Detect changes to a claim or exposure to see if it needs to be resent to ISO.
3. Construct an ISO message payload.

Due to data model changes, you probably need to customize ISO rules. ISO claim search requests can have various several types and each request type requires slightly different properties in the ISO XML message payload. Though the ISO destination has functions to create the different ISO claim search types, you need to tell ClaimCenter which combinations to use for a particular coverage or exposure. The ISO Gosu classes specify how to construct the appropriate message payload for an exposure and how to check that a particular exposure type is valid for ISO.

If you create new exposure types or change existing ones, you must change ISO payload generation rules to detect the new exposure type and generate appropriate XML for that subtype. If you fail to modify the payload, the payload generation functions returns `null` instead of an XML payload, and ClaimCenter sends nothing to ISO. You probably need to modify the injury, vehicle, property, or workers' compensation payload generation rules. In particular, the following functions in that library check for exposure types: `exposureFieldChanged` and `createSearchPayload`.

If you create new exposure types and do not customize ISO payload generation rules for this subtype, the ISO payload rules likely return null. This return value causes ClaimCenter to set the exposure's ISO status (the exposure's `ISOStatus` property) to the typecode for Not of interest to ISO.

ClaimCenter generates ISO payloads by using functions in the ISO library files in Studio. To locate these functions, in the **Project** window of Guidewire Studio, navigate to **configuration > grsc > libraries**, and then open **ISO**. This file contains functions that ClaimCenter calls to generate each type of ISO payload. However, this file is no longer the main location for payload generation code. The new payload generation system uses specialized classes. Each specialized class generates payloads for a certain type of ISO loss section. A loss section is the ISO term for a type of loss such as a property loss, a vehicle loss, and so forth.

To customize how ISO generates the payload XML for a loss section, modify the ClaimCenter class as defined in the following table.

Type of loss	ISO loss section	Customer class to modify in ClaimCenter package gw.api.iso
Auto loss	AutoLossInfo	ISOAutoLossSection
Information about injured people (standard, including third-party property)	ClaimsInjuredInfo	ISOInjuryLossSection
Information about injured people (Workers' Comp)	ClaimsInjuredInfo	ISOWCInjuryLossSection
Property loss insurance information for a person or insured party	PropertyLossInfo/ ClaimsSubjectInsuranceInfo	ISOPropertyLossSection
Property loss insurance information for an object	PropertyLossInfo/ItemInfo	ISOMobileEquipmentLossSection
Property loss insurance information for a water craft	PropertyLossInfo/Watercraft	ISOWatercraftLossSection

All the files in the previous table represent pairs of implementation classes. ClaimCenter organizes files in this way to simplify code merges for ISO payload generation during every upgrade.

For each section, the implementation class for your changes is the class listed in the previous table. However, these classes extend another base class that contains Guidewire core code for default behaviors.

This base file's file name has the suffix `Base`. For example, the `ISOPropertyLossSection` class extends from the `ISOPropertyLossSectionBase` class, which contains the bulk of the Guidewire code. If you want to see the existing implementation, refer to the base file. However, if you want to modify the behavior, Guidewire strongly recommends overriding methods in the non-base file rather than modifying the base class directly.

The code in the ISO library file determines which loss section class to use by getting the exposure enhancement property called `ISOLossSectionType`. The Gosu enhancement file `GWExposureISOEnhancement` implements this dynamic property. Customize that file if you want to change the mapping of exposure type to loss section type.

ISO payload generation properties reference

The full ISO hierarchy is described in the ISO documentation "XML User Manual DataPower v2.0." ISO strictly enforces this hierarchy. Request this manual from ISO and refer to it during customizations.

ISO prepare payload class

ClaimCenter calls the Gosu class `ISOPreparePayload` to add additional fields to a message immediately before ClaimCenter sends the message to ISO. Before calling this class, ClaimCenter ensures that it has already processed acknowledgments for any previous ISO messages related to the current claim. After `preparePayload` is called, the message is committed in the database.

The following two conditions are unknown at the time that Event Fired rules created the message. Messaging system functionality with database transactions ensures that they are known before the `ISOPreparePayload` class is called.

Message entity ID

The database ID property on the `Message` entity instance. This property is important because ClaimCenter uses it to construct a unique ISO request ID.

Known to ISO

All previous ISO messages are complete. Thus, the class can determine if the claim or exposure is known to ISO already. ClaimCenter uses this information to determine whether the message is an initial search or a replacement request for an item that ISO already knows about.

The purpose of the `preparePayload` method is to do the following tasks:

- Set the unique request ID for the request in the payload.

The request ID is a method argument that is based on the message ID, the claim/exposure ID, and the ISO customer password. The class sets this value for the `ClaimsSvcRq` element's `RqUID` field and the `ClaimInvestigationAddRq` element's `RqUID` field.

- Set the `ClaimInvestigationAddRq` element's `ReplacementInd` field.
 - If the claim or exposure is already known to ISO, the method sets the field to `1`.
 - If this claim search request is the first one, the method sets the field to `0`.
- Help ISO testing by reassigning IDs.

When testing ISO, it is common to run into problems. Tests typically send the same test data to ISO repeatedly, which does not work. The first time the test runs, the test data is not known to ISO, and ClaimCenter must send it as an initial request. However, the next time the test runs, ISO already knows about the test data, so the test must send only replacement requests. This requirement can make it hard to test the normal flow, which is to first send an initial message, and then to send a replacement message.

Therefore, when testing ISO integration, ClaimCenter uses the `ISO.properties` property `ISO.TestSuffix`.

- For production servers, always set this property to `0`.
- For testing, you can set this property to a unique positive integer. `ISOPreparePayload` uses this integer to append a unique suffix to identifiers in ISO requests, which causes ISO to treat the messages as different data during each test.

ISO match reports

A match report from ISO summarizes information about an exposure from another company that seems similar to an exposure in your system.

Match report as `ISOMatchReport` entity instances

After ISO receives a match report, ISO extracts selected properties from the ISO match report and associates it with a ClaimCenter claim or exposure. The ISO integration associates this information with the claim or exposure as an `ISOMatchReport` entity subtype. `ISOMatchReport` is a delegate that defines the interface for ISO match reports. The claim and exposure versions of ISO match reports implement this interface.

Both `ClaimISOMatchReport` and `ExposureISOMatchReport` entities contain the properties `ISOClaim` and `ISOExposure`. For `ClaimISOMatchReport`, the `ISOClaim` property references the claim and the `ISOExposure` property is `null`. For `ExposureISOMatchReport`, the `ISOExposure` property references the exposure and the `ISOClaim` property references the claim.

ClaimCenter stores match reports in the `ISOMatchReports` property on the claim or exposure entity. This property contains an array of one or more match reports, which are `ISOMatchReport` entity instances. You can customize how ISO match report properties populate properties on the `ISOMatchReport` entity.

Match report as an XML document

In addition to the `ISOMatchReport` objects, ClaimCenter saves the complete match report as a document on the claim in raw XML format. The name of the document is configurable and by default includes a date stamp. To change the document name, change the `ISO.properties` property called `ISO.MatchReportNameFormat`.

To display the match report, click the **ISO** tab on the claim or exposure. To display the document in the user interface, click the **Documents** tab on the claim or exposure. To get the match report document data, ClaimCenter requests the document from the document management system.

Each time you view a match report document, ClaimCenter renders the XML into HTML by using an XSL stylesheet that ISO provides. The XSL stylesheet reads raw XML and outputs the data in HTML. The document template file is called `ISOMatchReport.gosu.xml`. The document descriptor file is called `ISOMatchReport.gosu.xml.descriptor`. These two files implement the conversion of XML to HTML by using the XSL stylesheet.

The default template and descriptor work for most implementations. The only thing you might need to change is the context object for the URL of the stylesheet within in the descriptor file. You may need to update the URL to match the location of the XSL transform file that converts the XML to HTML.

```
<ContextObject name="xsl_file" type="string">
  <DefaultObjectValue>"http://SERVER:PORT/cc/resources/iso/xsl/CS_Xml_Output.xsl"</DefaultObjectValue>
</ContextObject>
```

Change the URL to change to match the actual URL of ClaimCenter server or your proxy server.

The match report XML contains only supported ISO properties and does not include any custom extensions. Only update the XSL file if ISO changes their match report XML format to provide additional (or different) data in the future.

If you customize a ISO match report code, you must modify the code that creates a new match report. From a claim or an exposure, to create the correct type of match report subtype, call the `ISOReportable.addNewISOMatchReport()` method. Both claim and exposure implement the `ISOReportable` interface, so this method is available from Gosu on `Claim` and `Exposure` objects.

To customize how ClaimCenter creates new ISO match reports from XML, modify the class in the method `populateMatchReportFromXML` in the class `ISOReply`. If you override this method, remember to call the superclass version first.

Any changes to the match reports take effect for the next generated ISO Match document that ClaimCenter creates when it receives the next ISO response.

ISO exposure type changes

If you enable ISO, by default ClaimCenter sends exposures of the following types to ISO for fraud detection: `BodilyInjuryDamage`, `VehicleDamage`, `PropertyDamage`, `LossOfUseDamage`, and `WCInjuryDamage`. The `WCInjuryDamage` type covers all workers' compensation claims, so there is no need to also submit `TimeLoss` exposures for those exposures.

Sending additional exposure types to ISO requires several types of changes. Be sure to allocate sufficient engineering and testing time to the following integration elements.

Task	Description
Update exposure type to loss section mapping	Update the exposure enhancement file <code>GWExposureISOEnhancement</code> to customize the <code>ISOLossSectionType</code> enhancement property, which defines the mapping of exposure type to loss section mapping Gosu class.
Update ISO payload rules	You must change ISO payload generation rules to detect the new exposure type and generate XML appropriate for that subtype. If you fail to modify the payload, the payload generation functions return null instead of an XML payload, and nothing sends to ISO. You probably need to modify the injury, vehicle,

Task	Description
	property, or workers compensation payload generation rules. Review the loss section class files to understand or customize the logic for each loss section.
Update ISO user interface	If you use exposure-level messaging only, change the PCF page definition (user interface definition) for the Exposure Details view of that type to add the ISO responses subpage. If you use claim-level messaging only, you do not need this step. However, if you have legacy exposures that are known to ISO, you must perform this step so that the user interface is appropriate for exposures in those special cases.
Update typecode mapping	Update the typecode mapping file with your data model changes.
Update coverage mapping	Update the coverage mapping file with your data model changes.
Update validation rules	Update your validation rules to include new rules for any required properties on your exposure types.

If the payload generation functions return `null`, ClaimCenter displays the ISO status as “Not of interest to ISO.” Returning `null` typically is caused by an exposure type that is a custom subtype not handled by the default ISO payload generation rules.

ISO date search range and resubmitting exposures

After a claim is entered in the ISO database, ISO searches for claim matches for a well-defined time period.

This period is 30 days for auto insurance, 60 days for property insurance, and 1 year for casualty insurance. There is no way either to extend that time period or to automatically resubmit the claim for a longer total search period.

Based on the initial received date, ISO stops sending automatic update reports after the applicable time period has elapsed. Sending additional updates on that claim cannot extend the automatic update window. Do not attempt to resubmit claims or exposures to ISO for this purpose.

ISO integration troubleshooting

If your computer is not getting responses back from ISO, first start logging and message logging.

Enable the following logs.

- To start ClaimCenter logging, set category `log4j.category.Integration.messaging.ISO` to `DEBUG` in `log4j2.xml`.
- To start message logging, set `ISO.LogMessagesDir` to a valid directory in your `ISO.properties`.
- On your proxy server, enable all logs for the forwarding application on the server such as Apache or Squid. Refer to the application documentation for details on this process.
- On your firewall, enable all logs.
- On your proxy server, enable any other logs, such as general system logs.

Once message logging is on, you can see which messages sent, whether you received receipts, and whether you received responses.

If there were no ISO requests, check if the exposure ISO status is not “Not of interest.” If it does have that status, check the ISO rules to see if the rules handle this type of exposure. If they do not handle the custom exposure type, and thus return `null` for the ISO payload, the exposure gets this ISO status. You can reset the ISO status property by logging in as an administrator and viewing the exposure in the **Exposure Detail** page.

If you see a request but did not receive a receipt, the message may not have been sent to ISO.

Confirm all of the following conditions.

- Check that the ISO.SendMessages property in ISO.properties is set to true.
- Check that the ISO.ConnectionURL property in ISO.properties is set to the proxy server URL. (This confirmation assumes that you are using a proxy server, which is the recommended setup.)
- Check that the ISO server is up. Test the ISO server URL in your web browser.
- If you see a receipt, open it and see if it contains the correct MsgStatusCd value ResponsePending. If it does not, look at the error message. Probably the authentication information in ISO.properties is incorrect. Possibly you are sending the request from an IP address outside the range registered with ISO.
- If the receipt looks good but you do not see a response then check that your proxy server is active and set up correctly. Check if the request arrived at the proxy server and check the logs of the proxy server, such as Apache or Squid. If not, the ISO server may be down, or the bastion host/forwarder is not listening on the correct URL. If the request arrived at the forwarder, check the connection between the forwarder and your servers that can execute the messaging operation. Carefully check your proxy server configuration files.
- If you see a response but it contains an error code, look at the associated error message. Most errors are configuration problems such as missing required properties or incorrect policy/coverage/loss types. Sometimes the ISO server thinks an exposure is a duplicate of an exposure that it knows already. Or, ISO does not know about an exposure that ClaimCenter thinks it knows. In these cases, login as an administrator and change the **ISO Known** flag on the **Exposure Detail** page.

Although the Administration pages for messages are not ISO-specific, outgoing requests to ISO are handled by using the messaging system. It is sometimes necessary for ClaimCenter administrators to track connectivity issues by using this user interface.

ISO-specific error codes

The most up to date and complete list of ISO errors is in the ISO documentation called the “ISO XML User Manual.”

For further detail on errors returned from ISO, contact your ISO customer support representative. ISO’s customer support can help you learn about the error. Encourage them to add additional new error information to the ISO documentation as needed.

Testing automatic updates

After ClaimCenter sends an exposure to ISO and ISO responds, ISO keeps the exposure in its database. Afterwards, as other companies send claims to ISO, ISO might detect matches to one of your exposures. If a match is detected within a certain time range, ISO sends a match report to the ClaimCenter callback URL. The time ranges are 1 year for casualty, 60 days for property and 30 days for auto. ISO refers to these delayed responses as automatic updates.

Always test your system’s responses to automatic updates, but only after normal ISO receipts and responses are working reliably. However, testing automatic updates is difficult because automatic updates are sent only after another company adds a matching claim, so you cannot cause an automatic update yourself. To test automatic updates, contact your ISO representative. They can manually submit a matching claim marked with a different (test) customer ID.

ISO formats and feeds

You can access the ISO Claim Search service by using two different format types. ISO uses the same database for all protocols and feed methods. However, the data format is different within the ISO database for the universal format compared to the legacy FTP format.

ISO ClaimSearch universal format

ISO’s universal format supports multiple types of claims, such as casualty claims, property claims, and auto claims. You can choose one of three different feeds types which indicate basic transport types.

- **Universal format XML feed** – Upon creating or changing a claim, a claims management system sends an XML-formatted message to ISO, which ISO translates to and from universal format. Matches are sent back asynchronously, but immediately upon detection of the match. The XML messages can be submitted as a simple

HTTP POST request or as a HTTP SOAP web service request. However, ClaimCenter requires the SOAP version for initial queries. Replies from these ISO requests are always in universal format XML feed HTTP POST format. Notify ISO about a related configuration setting to ensure that ISO is expecting the SOAP request.

- **Universal format FTP feed** – Daily claims are aggregated in one flat file sent daily as a batch file to ISO. ISO sends matches back as batch results only once per day, in contrast to the immediate (although asynchronous) response of the XML feed type.
- **Universal format MQ feed** – ISO supports sending and receiving by using the WebSphere MQ messaging system.

ISO legacy FTP format

Prior to ISO's universal format, ISO required claims in legacy mono-line FTP formats provided by INDEX (casualty insurance), PILR (property insurance), and NICB (auto insurance). Like the universal format FTP feed, daily claims are aggregated into one flat file which is uploaded daily to ISO by using FTP.

Although the database is the same for all access formats and feed types, the legacy FTP format uses a different data format within the ISO database. Be careful not to confuse legacy FTP format with the universal format access with FTP feed.

ISO web interface

ISO provides a website to enter claims manually. You can log on at a later time to view found matches from queries you enter manually or queries you submit by using other feeds. However, the HTML web interface is not a separate data format type or feed type.

Migrating old claims systems to ClaimCenter

If ClaimCenter is replacing a claim management system that already feeds the legacy ISO databases, all legacy claims must be converted to the ISO Master Claims Database. The basic process is to resubmit the claim in universal format by using the identical claim number using the conversion tag. This resubmission prompts a process at ISO that converts the legacy record to universal format. After that you can submit updates by using standard processes.

If you need to change any information such as converting claim numbers to the format expected by ClaimCenter, you can send a key field update transaction. Custom logic can be put in to trigger this transaction. Remember also that the old system must stop sending claims as soon as the new system goes live to prevent duplicate records in ISO's production database.

You must ensure that the old system stops sending claims with the old system as soon as the new system goes live to prevent duplicate records in ISO's production database.

If ClaimCenter is replacing a claim management system that uses the universal format FTP feed, claims need no conversion because the original claims are already stored in the universal format. Therefore, ClaimCenter deployment for this type of migration is relatively straightforward with respect to ISO data.

As mentioned earlier in this topic, ISO supports the following input methods at the same time.

- One system with universal format FTP feed using the *production database*
- Another system using the universal format XML feed with the *testing database*

This feature makes it easy to migrate from universal format FTP feed to the universal format web service feed.

If the claim was never sent to ISO in the legacy system, ISO was unaware of it. There is no need to run the conversion process for that claim. It can proceed normally in ClaimCenter.

Update the XML files provided by ISO

About this task

ISO sometimes changes their XML schema to support new features. These changes are almost always additive. If they are not additive, ISO makes a large number of notifications before release. If the change are additive and changes

happen outside the ClaimCenter product release cycle, you can update the XML files and associated source code to support these changes.

Procedure

1. Get the updated XSD files from ISO. When ISO releases a new version, ISO supplies customers with three XSD files that are stored in the package `xsd.iso`.
2. Use a difference detection tool to compare the ISO files with the XSD files in ClaimCenter.

The ClaimCenter files are in the directory `ClaimCenter/modules/configuration/generated/xsd/isoxsd/iso`.

- `req.xsd` is for requests.
- `ak.xsd` is for acknowledgments.
- `resp.xsd` is for responses.

In the default configuration, ClaimCenter only uses `resp.xsd`.

Note: In Guidewire Studio, you can see these files by navigating in the **Project** window to **configuration > gsrc > xsd > iso**.

3. In the ClaimCenter versions of the file, make the same changes ISO has made.
4. Write new Gosu code that enables you to set the new fields.

Metropolitan Reporting Bureau integration

Metropolitan Reporting Bureau provides a nationwide police accident and incident report service in the United States. Many insurers use this system to obtain police accident and incident reports to improve record keeping and to reduce fraud. ClaimCenter support for the service decreases deployment time for Metropolitan Reporting Bureau integration projects, particularly for personal lines insurers.

See also

- For more information about the services that the Metropolitan Reporting Bureau provides, refer to the bureau's website.
<http://www.metroreporting.com>

Overview of ClaimCenter integration with Metropolitan Reporting Bureau

ClaimCenter integrates with Metropolitan Reporting Bureau (Metropolitan) by using their XML Gateway for requesting police accident and incident reports. You enter all the necessary data in ClaimCenter and then send the information through the gateway. You do not need to visit the Metropolitan web site to request reports or to view reports.

Metropolitan integration includes the following features.

Ordering a report during claim intake

An adjuster or customer service representative on the phone with an insured customer is taking in a First Notice of Loss (FNOL) report through the ClaimCenter New Claim Wizard. The adjuster can also submit a request for a police report.

Ordering a report on an established claim

If a police report was not requested originally during claim intake, or First Notice of Loss (FNOL), the adjuster can order one later from the claim file user interface.

Multiple reports on the same claim

Sometimes an adjuster requests a police report for a claim but has some data incorrect, such as the police department details. An adjuster can change the appropriate information and submits a request for another, new report.

Many report types

Metropolitan has approximately 30 report types. ClaimCenter supports most of the report types.

Attaching a report to a claim file as a document

After adjusters request reports, the reports are retrieved later, asynchronously. After Metropolitan returns the report, ClaimCenter matches it to a specific claim and attaches it as a document to the claim file. Users can view or print the report like any other document.

Report completion notification

ClaimCenter notifies the adjuster assigned to a claim if a requested report successfully attaches to the claim file and is available for review. Alternatively, if the report request fails for some reason, ClaimCenter notifies the adjuster. This notification is implemented as an activity.

There are two ways to view the **Metropolitan Reports** detail page in ClaimCenter. Both require that you open a claim and click **Loss Details** in the Sidebar to open the screen. After the **Loss Details** screen is open, the following actions can be performed.

- If reports have been received, you can scroll to the **Metropolitan Reports** section and click the link in the **Type** column.
- If reports have not been received or you want to add new ones, click **Edit** to put the **Loss Details** page in edit mode. Then scroll to the **Metropolitan Reports** section and click the **Add** button.

The Metropolitan Reports list view has the following columns:

Type

The first column in the list view is the type of the report request. After you click on the report type link, ClaimCenter displays a page with details about this report request.

Status

The current status of the report request.

Order date

The date that ClaimCenter sent the report request to Metropolitan.

Document

The document column shows the name of the document if the report status is Received. The user can view the document by clicking the **View Document** button in the **Actions** column.

Actions (View Document, Resubmit)

The last column is for buttons to view a document or to resubmit the report request. If the original report request has insufficient data, you can update the claim with all required properties. You can return to the loss detail page and click the **Resubmit** button to rerun preupdate rules. ClaimCenter can send a new report request if it passes preupdate rules.

In the **Metropolitan Report** list, the user can click the report type to open the details screen and then enter any required data for that report. Rather than replicate the Metropolitan form for every report type, ClaimCenter extracts relevant data from entities on the claim as appropriate. ClaimCenter copies this information to a new **MetroReport** entity in the property `claim.MetroReports` to track the report.

After a user requests a Metropolitan report, ClaimCenter returns to the **Loss Details** page. You see a new row in the Metropolitan reports list with a report **Status** of New.

The following table shows the current report types and their report type codes from the **MetroReportType** typelist.

Request code	Request description	Request code	Request description
A	Auto Accident	N	OSHA
B	Auto Theft	O	Other

Request code	Request description	Request code	Request description
C	Auto Theft Recovery	P	Activities Fixed Rate
D	Driving History	Q	Property and Judgements
E	Coroner Reports	R	Registration Check/DMV
F	Fire - Home	S	Insurance Check
G	Burglary	T	Title History
H	Death Certificate	U	Subrogation Financial/Assets
I	Incident	V	Vandalism - Auto
J	Locate Defendant/Witness	W	Weather Report
K	EMS/Rescue Squad	X	Fire - Auto
L	Supplemental/Addendum	Y	Photos
M	MV-140 (NY Only)	Z	Disposition of Charges

ClaimCenter Metropolitan integration architecture

ClaimCenter integrates with Metropolitan Reporting Bureau with the following types of code, only some of which are directly modifiable or configurable:

Messaging plugins

Messaging plugins are able to send outgoing messages to Metropolitan. These messaging plugins repeatedly try to send the request to the Metropolitan servers until it acknowledges the request. An acknowledgment of the request indicates receipt of the request, not that the report is ready yet. A different part of ClaimCenter queries Metropolitan regularly to determine if the report is ready. These plugins are minimally configurable through configuration files.

Preupdate rules

Claim preupdate rules verify that all the required data is available for a claim. You can modify these rules.

Report templates

ClaimCenter provides Gosu templates that generate XML data to send to Metropolitan, based on ClaimCenter claim data. You can modify these templates.

Event-handling rules

ClaimCenter provides event handling rules for Metropolitan internal messaging events. The messaging events send messages to the Metropolitan messaging plugins to handle outgoing transport and asynchronous replies. These event-handling rules are part of the base implementation. You cannot modify these event-handling rules.

Documents support

After Metropolitan's APIs indicate that a report is available, Metropolitan provides a URL to download the report. ClaimCenter support for documents enables police reports and other reports to be stored as documents in ClaimCenter in the claim file. In the base configuration of ClaimCenter, Metropolitan messaging plugins use document management APIs to add the document to ClaimCenter.

Internal state machine that manages Metropolitan report status flow

In the base configuration of ClaimCenter, the Metropolitan integration includes an internal state machine that manages flow of Metropolitan report status. This state machine is not modifiable. This state machine queries Metropolitan regularly using Metropolitan's APIs to determine if a report is ready.

IMPORTANT: Because these requests happen over the Internet, you must provide appropriate firewall protection for requests, and appropriate firewall holes to permit communication with Metropolitan.

All communications with Metropolitan are initiated by ClaimCenter. After the original report request, ClaimCenter polls Metropolitan regularly. The consequence is that it is easier to configure network proxies since all requests are HTTP requests outgoing from ClaimCenter to Metropolitan, and a reverse proxy is not necessary. However, insulating ClaimCenter outgoing requests through a proxy is still a good network practice.

See also

- “Metropolitan report statuses and workflow” on page 625

Metropolitan configuration

There are three main places where you must configure Metropolitan:

1. The main application configuration file, `config.xml`. See “Enable Metropolitan configuration” on page 620.
2. The Metropolitan properties file, `Metro.properties`. See “Metropolitan properties file” on page 620.
3. Display strings in the `Metro` section of the **Localization** keys. See “ClaimCenter display keys for Metropolitan reports” on page 621.

Enable Metropolitan configuration

Metropolitan integration can be enabled or disabled by changing one parameter in the main application configuration file, `messaging-config.xml`. The corresponding Boolean parameter is defined as shown in the following configuration definition.

```
<param name="EnableMetroIntegration" value="true"/>
```

Metropolitan-related messaging plugins are defined in `messaging-config.xml`. You can make minor configuration changes, such as retry times, to these destination settings.

Metropolitan properties file

The Metropolitan properties file `Metro.properties` contains most of the configuration information for the Metropolitan integration points.

Because the report requests are not pushed to ClaimCenter, ClaimCenter must poll the Metropolitan server regularly. One of the most important configuration parameters that you can make is the time interval between inquiries. This interval is effectively the delay for a polling request to the server to see if a report is available. The property `Metro.InquiryInterval` specifies the number of minutes to wait between requests, with a minimum of 60 minutes. The default setting in the base configuration is **150**.

```
Metro.InquiryInterval = 150
```

Additionally, update the following properties specifically for each client with the basic ClaimCenter customer information. Properties include `CustAccount` and `CustBillingAccount`, which are account and billing numbers for each Metropolitan account so the client is charged appropriately. Metropolitan provides these numbers for each client.

The `CustNAIC` property is the NAIC number for the ordering company. If you do not know this number, put the value `NONE`.

The following settings are examples of the base configuration properties in the `Metro.properties` file.

```

Metro.CustNAIC      = 11882
Metro.CustCompanyName = Allstate Insurance
Metro.CustAddress1   = 123 West Ave
Metro.CustAddress2   =
Metro.CustCity       = Erie
Metro.CustState      = PA
Metro.CustZip        = 41222
Metro.CustAccount    = PING
Metro.CustBillingAccount= MRDEMO

```

There are other properties in that file that typically do not require modification. Change the other properties only if Metropolitan changes the XML formats for date, time, and social security numbers. These property formats are in standard regular expression format.

Credentials management

The `Metro.LoginId` and `Metro.Pswd` credentials are set by an implementation of the `CredentialsPlugin`, the `SecretsManagerCredentialsPlugin.gs` class.

Each credential value (also known as a *secret*) is a combination of a username and password and follows the `login:password` syntax.

Note: You may also use an external substitution file in ClaimCenter to set credentials. See "External server configuration" in the *Administration Guide*.

IMPORTANT: Do not enter the `Metro.LoginId` and `Metro.Pswd` credentials directly in the `Metro.properties` file as this method of storing credentials is not secure.

ClaimCenter display keys for Metropolitan reports

In Guidewire Studio for ClaimCenter, you can set display keys to define your own error messages, property names, and other text to display for Metropolitan Reports.

To modify display keys, in the **Project** window, navigate to **configuration > config > Localizations > Resource Bundle 'display'**, and then open the display key property file for your locale.

For example, in the base configuration the file for U.S. English is `display.properties`. Navigate in the **Project** window to **configuration > config > Localizations > Resource Bundle 'display'** and double-click `display.properties` to open this file in the editor.

You can modify any of the Metropolitan Report display keys by searching for `metro`, selecting a display key, and editing it. Some display keys with their default settings are listed below.

- `Metro.Activity.InsufficientData.Message` – Default setting is `Missing field(s)\:`
- `Metro.Fields.AgentCity` – Default setting is `City of the investigating agency`.
- `Metro.Fields.ClaimNumber` – Default setting is `{Term.Claim.Proper} Number`.

The Metropolitan Reports error messages and property names are used in preupdate rules if there are required properties that are not defined. ClaimCenter creates an activity that indicates the missing properties so you know where to fix the report request, after which you can resubmit the request.

There are many other display keys that include `metro` that you can modify.

Configure activity patterns

The Metropolitan integration includes the following activity patterns. An activity pattern is a type of template for a user activity notification. If you log in to ClaimCenter as an administrator, you can modify these activity patterns by clicking the **Administration** tab and clicking **Activity Patterns** in the sidebar menu.

The Metropolitan activity patterns are described below.

- **Metropolitan Report Available** – If a report request succeeds and the report is attached to the claim, ClaimCenter creates a `metropolitan_report_available` activity. ClaimCenter assigns it to the user that created the `MetroReport` entity associated with the request. The Metropolitan workflow uses this activity pattern.
- **Metropolitan Report Deferred** – For report requests that takes additional time to process, Metropolitan replies with the response `deferred`. ClaimCenter creates the activity `metropolitan_report_deferred`. The Metropolitan workflow uses this activity pattern.
- **Metropolitan Report Held** – If Metropolitan needs additional information from ClaimCenter before Metropolitan can process it, Metropolitan replies with the response `hold`. In response, ClaimCenter creates the activity `metropolitan_report_held`. The Metropolitan workflow uses this activity pattern.

- **Metropolitan Report Inquiry Failed** – Used if a request to Metropolitan other than the initial order request fails. A `metropolitan_report_inquiry_failed` activity is generated and then the workflow retries. The Metropolitan workflow uses this activity pattern.
- **Metropolitan Report Request Failed** – For any report request that fails due to incomplete data or if initial order message fails. This activity pattern creates a `metropolitan_request_failed` activity and assigns it to the user that created the `MetroReport` entity associated with the request. The activity description text area includes the type of report requested and the data that must be supplied to successfully submit the request. The Metropolitan workflow and the `Metro.gs` library uses this activity pattern.
- **No Metropolitan Report Available** – For any report request that fails after sending the request because no report is available. ClaimCenter creates a `metropolitan_report_unavailable` activity and assigns it to the user that created the `MetroReport` entity associated with the request. The Metropolitan workflow uses this activity pattern.

Configure the messaging plugin retries

About this task

If the messaging plugins fail to send to Metropolitan, they retry after a specified time interval up to a maximum number of retries. If Metropolitan does not acknowledge receipt successfully, the messaging plugins retry at the specified time interval up to the maximum number of retries. To change the default number of retries to attempt, perform the following operations.

Procedure

1. In the Project window in Guidewire Studio, navigate to **configuration > config > Messaging**, and open the `messaging-config.xml` file.
2. In the list on the left, click the row for ID 67 and the name `Java.MessageDestination.Metro.Name`. This message destination is built-in and cannot be modified other than through its configuration settings.
3. Edit the following values.
 - **Max Retries** – Default value is 5.
 - **Initial Retry Interval** – Default value is 1000 milliseconds.

Metropolitan report templates and report types

ClaimCenter report requests to Metropolitan must be formatted in the correct XML format for that report type. To generate this request, ClaimCenter runs a specific a Gosu template, which is a text file with embedded Gosu code. This template generates the necessary XML-formatted text.

In this topic:

- “Metro report types and loss types” on page 622
- “Edit Gosu template files” on page 623
- “Map report types to Gosu template files” on page 623
- “Metro report template special cases” on page 623
- “Add new report types” on page 624

Metro report types and loss types

ClaimCenter includes templates for all report types, and these report types are auto-configured to correspond to the standard loss types in the ClaimCenter reference configuration. For example, Auto Accident reports map to the Auto loss type, and Coroner Report maps to all base loss types. The typelist `MetroReportType.ttx` specifies the mapping between a Metropolitan report type and a ClaimCenter loss type. You can change this mapping, but be aware that Metropolitan specifies different data requirements and constraints for each report.

Different report types have different data requirements. Gosu templates and preupdate rules determine if the required data for the selected report type was correctly specified by the adjuster.

Edit Gosu template files

You can customize report templates to support your own changes to the ClaimCenter data model. Also, you might need to customize reports if, at a later date, Metropolitan changes the XML format, such as changing tag names or adding required properties for certain report types. You can modify Metropolitan report Gosu templates, but do not change template headers or footers.

Review the latest version of the Metropolitan specification at the following URL.

```
https://metroweb.metroreporting.com/schema/index.php
```

The Metropolitan report Gosu templates are stored, one template per report, in the following directory.

```
ClaimCenter/modules/configuration/config/web/templates/metroreport
```

For example, the `BurglaryReport.gst` template generates the Metropolitan burglary report request.

A sample report template looks like the following text.

```
<mrb:Insured>
  <mrb:CompanyName><%=claim.Insured.Company.Name%></mrb:CompanyName>
<% } else { %>
  <mrb:Last><%=claim.Insured.Person.LastName%></mrb:Last>
  <mrb:First><%=claim.Insured.Person.FirstName%></mrb:First>
  <mrb:Middle><=%=translator.formatNullToEmptyString(claim.Insured.Person.MiddleName)%></mrb:Middle>
<% } %>
  <mrb:Address1><=%=translator.formatNullToEmptyString(claim.Insured.PrimaryAddress.AddressLine1)%>
  </mrb:Address1>
  <mrb:Address2><=%=translator.formatNullToEmptyString(claim.Insured.PrimaryAddress.AddressLine2)%>
  </mrb:Address2>
  <mrb:City><=%=translator.formatNullToEmptyString(claim.Insured.PrimaryAddress.City)%></mrb:City>
  <mrb:State><=%=translator.formatNullToEmptyString(claim.Insured.PrimaryAddress.State)%></mrb:State>
  <mrb:Zip><=%=translator.formatNullToEmptyString(claim.Insured.PrimaryAddress.PostalCode)%></mrb:Zip>
  <mrb:Phone><=%=translator.formatPhoneNumber(claim.Insured.PrimaryPhoneValue)%></mrb:Phone>
</mrb:Insured>
```

As the example template shows, the templates use `translator` classes to format the data by using an object with the symbol `translator`. This utility class enables report templates to use a Metropolitan utility class that translates strings in report templates with the following methods.

- `formatCustDate` – Format date to MMDDYYYY format, which is the required Metropolitan date format.
- `formatCustTime` – Format time to HHMM 24 hours clock format, which is the required time format.
- `formatPhoneNumber` – Format the phone number to 10 digits without spaces or other characters.
- `formatNullToEmptyString` – Translate a null object reference to an empty string. This method is particularly useful for optional properties in XML templates.

Map report types to Gosu template files

The mapping between template files and the report types is specified by the XML configuration file `MetroReportTemplateMapping.xml`. Your Gosu template file names must all end in `.gst`. The following example includes one `<report>` element that maps the Metropolitan report A (auto accident report) to a specific Gosu template file.

```
<Report>
  <Type>A</Type>
  <Template>MetroAutoAccidentReport.gst</Template>
</Report>
```

Metro report template special cases

ClaimCenter handles claims insured by companies, auto accident reports with no drivers, and workers' compensation as follows for Metropolitan reports.

Claims insured by companies

For a claim insured by a company, ClaimCenter puts the company's information in <insured>.

Auto accident reports and null drivers

For the Auto Accident report type, if the specified driver is null, ClaimCenter puts Parked in the driver's first name and in the last name within the report request. The assumption is that, at the time the auto accident happened, the car was parked and there was no driver inside.

Workers' compensation

In the ClaimCenter base configuration, workers' compensation claims do not include all the requisite data that Metropolitan requires for reports. For workers' compensation reports, ClaimCenter puts the claimant's information in the <driver> property because there is no direct mapping from the ClaimCenter data model to the Metropolitan XML data document.

Consequently, workers' compensation claims cannot generate some reports. The user interface automatically removes unsupported report options if the claim is a workers' compensation claim.

Add new report types

About this task

When Metropolitan makes new report types available, they can be added to ClaimCenter.

Procedure

1. Create a report template for the new report type.
2. Save the template in the following directory.

```
ClaimCenter/modules/configuration/config/web/templates/metroreport
```
3. Modify the file `MetroReportTemplateMapping.xml` to add mapping information for the new report type.
4. Add the new type to the report type mapping file `MetroReportType.ttx`.
5. Update the ClaimCenter preupdate rules to check for all required properties for the new report type.
6. Regenerate Java API and SOAP API files.
7. Rebuild and redeploy the application WAR file to make the new mappings and files available at runtime.

See also

- “Regenerating integration libraries and WSDL” on page 33

Metropolitan entities, typelists, properties, and statuses

In this topic:

- “MetroReport entity” on page 625
- “MetroReportType typelist” on page 625
- “MetroAgencyType typelist” on page 625
- “Document entity” on page 625
- “Metropolitan report statuses and workflow” on page 625
- “Customizing Metropolitan timeouts” on page 627

MetroReport entity

The entity MetroReport describes one report from Metropolitan, both before a response is returned and afterward. There is an array of MetroReport objects in a MetroReports property on Claim.

MetroReportType typelist

Each MetroReport entity has an associated MetroReportType. The MetroReportType typelist defines the possible types of Metropolitan reports adjusters can request, such as Auto Accident, Auto Theft, Coroner Reports, Title History, and others. The MetroReportType typelist also maps Metropolitan report types to loss types. For example, an adjuster can request an Auto Accident type of report only if the loss type on the claim is Auto.

The following example shows a MetroReportType typecode for a burglary report, which adjusters can request only if the lost type is Auto or PR.

```
<typecode code="G" name="Burglary" desc="Burglary" priority="7">
  <category typelist="LossType" code="AUTO"/>
  <category typelist="LossType" code="PR"/>
</typecode>
```

MetroAgencyType typelist

Each MetroReport entity has an associated MetroAgencyType. The MetroAgencyType typelist defines the possible investigating agency types for a report. Users who request reports from the user interface can specify the investigating agency type. If a requester does not have the information, the MetroAgencyType property can be null.

The following examples show MetroAgencyType typecodes for investigating agency types, with codes, names and descriptions.

```
<typecode code="PD" name="Police Department" desc="Police Department" priority="1"/>
<typecode code="CO_PD" name="County Police" desc="County Police" priority="2"/>
```

Document entity

Each MetroReport entity has a foreign key reference to a Document entity, which is the document generated by Metropolitan if a report request results in an actual report.

If Metropolitan successfully returns a report, the built-in code requests that the document content source plugin implementation registered in `IDocumentContentSource.gwp` add the document to the claim.

Metropolitan report statuses and workflow

Each MetroReport entity has a MetroReportStatus property that indicates the current status of the report request. The following list of report statuses suggests the flow of the report request process, as does the diagram of Metropolitan workflow states and activities after this list. This list of statuses also helps you understand the meaning of user-visible text in the application user interface.

none

No order was located that matched the information supplied.

accepted

The order was accepted by the Metropolitan Reporting Bureau.

closed

The order was canceled, or it cannot be completed for some reason.

deferred

An image was returned with a notice that the data source takes some additional time to provide the requested information. The order is still open.

downloadingreport

The system is in the process of downloading the report.

duplicate

This typecode is not used. A duplicate report request initially has the status Pending, In Progress. ClaimCenter queries the status of the original request and, if successful, assigns that request status to the duplicate request.

error

The inquiry sent could not be matched to any existing order in the Metropolitan Reporting Bureau system.

hasreport

The report is ready on the external server.

hold

The order is awaiting additional information from you.

inquiryfailed

The inquiry request failed based on the result sent back from the Metropolitan Reporting Bureau.

insufficientdata

Some of the required fields are missing.

new

The initial status of the report request.

orderfailed

The order request failed based on the result sent back from the Metropolitan Reporting Bureau.

pending

The order is in the system and is currently awaiting a response from the data source.

received

The Report has been received—downloaded successfully—to your server.

sendinginquiry

The report inquiry file has been sent and is waiting for the response from the Metropolitan Reporting Bureau.

sendingorder

The order has been sent and is waiting for the response from the Metropolitan Reporting Bureau.

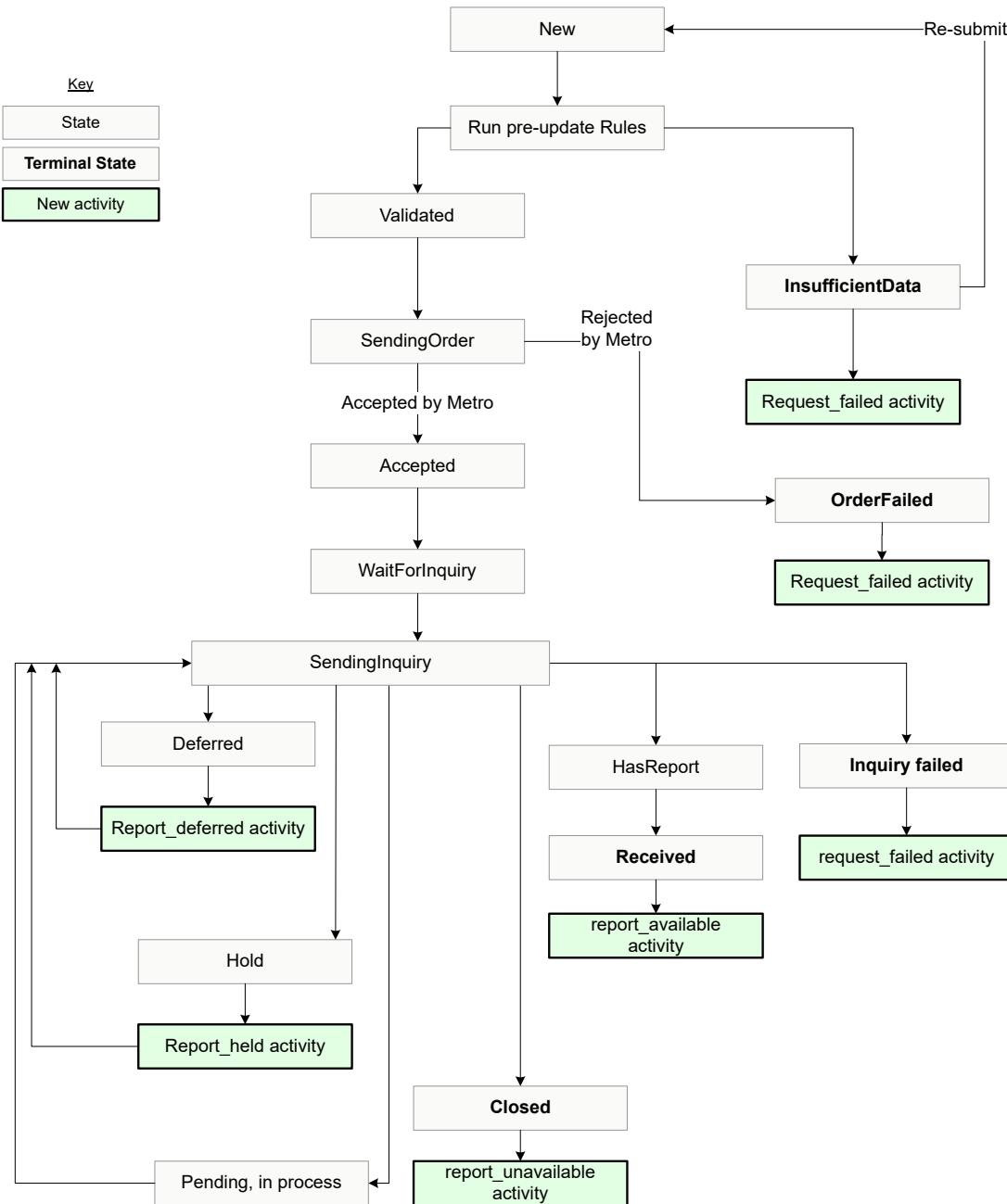
validated

The request has been validated and ready to be sent out.

The process for requesting Metropolitan reports and the statuses through which the process flows are fixed in ClaimCenter. You cannot modify the process flow, nor can you modify or extend the set of process statuses. The only ways to configure the Metropolitan report request process are by setting the `InquiryInterval` parameter and other settings documented in this topic.

The following diagram illustrates the Metropolitan report request process and the changes in status that can occur for a specific report request. The diagram also shows the points in the process at which ClaimCenter generates user activities related to a specific report request.

Metropolitan Workflow States and Activities



Customizing Metropolitan timeouts

The built-in integration of Metropolitan reports has two time cycles in the workflow.

- A delay after sending the request before requesting the report.
- A delay after asking for the report, if the Metropolitan server does not have the report before checking again if the report is ready.

Both time cycles have associated time values in the workflow. The defaults are described below.

- **Metro.InquiryInterval** – After submitting the original request, ClaimCenter waits 2.5 hours before inquiring initially. You can modify this value in the `metro.properties` file.
- **Metro.OrderTimeout** – After requesting the report, ClaimCenter waits for the response 8 hours. You can modify this timeout in the `metro.properties`. This is exposed in Gosu as the property `metroReport.PastOrderTimeout`. It returns true only if the request started and has been waiting longer than the corresponding timeout value.

Additionally, if the overall workflow does not finish completely within one week, then ClaimCenter stops it and creates an error activity. You can modify this timeout in the `metro.properties` file in the property `Metro.WorkflowTimeout`. This is exposed in Gosu as the `MetroReport` property `metroReport.PastWorkflowTimeout`. It returns true only if the workflow started and has been running longer than the corresponding workflow timeout value.

In the `metro.properties` file, specify the time values for the two delays and the timeout using a number followed by a suffix for the unit. Use `d` for days, `h` for hours, and `m` for minutes. For example, use `2d` for “two days.” Use a combination of values, such as `3d2h`, for “three days and two hours.”

Metropolitan error handling

The following considerations affect error handling with Metropolitan Reporting Bureau:

- Preupdate rules must ensure that all required data for each report type is specified before a report request is sent to Metropolitan.
- If Metropolitan denies the report request, ClaimCenter updates the report status and creates a new activity for the adjuster to examine and resend the request if appropriate.
- The messaging plugins can be configured to retry sending to Metropolitan until a maximum number of times is reached. If no acknowledgment returns from Metropolitan, the messaging plugins retry at the specified time interval up to the maximum number of retries.
- Metropolitan sometimes changes the format of the XML files that Metropolitan requires for requests, and it sometimes adds required properties. Review the latest version of the XML specifications at the following web address.

<https://metroweb.metroreporting.com/schema/index.php>

Data transfer structures

ClaimCenter supports multiple formats for transferring data to and from external systems. Typical formats include JSON and XML structures. For bulk data transfer, ClaimCenter supports using files on the local file system or in an Amazon S3 bucket.

Uses of data transfer structures include integration to an archival repository for records that no longer need immediate access on a live system.

Guidewire InsurancePlatform Integration Views

Integration Views supports selecting and retrieving data from the ClaimCenter database and further composing and transforming that data in a custom manner to meet business needs. With Integration Views, you define a stable, versioned contract for this data processing. Integration Views provides a straightforward, declarative way to map underlying ClaimCenter data into a serialized format that conforms to the contract.

You define the contract explicitly and independently from the underlying ClaimCenter data model. In this way, the contract is decoupled from the data model. When you change the contract, the change is both explicit and intentional.

Integration Views supports the coexistence of different versions of the contract so that you can evolve the contract and the systems that rely upon it in a controlled fashion to meet business needs.

Simultaneously, Guidewire can improve the ClaimCenter data model. By decoupling your data integration process from the Guidewire data modeling process, Integration Views makes it possible for you and Guidewire to work together without interference to achieve common progress.

Overview of Integration Views

The essence of Integration Views is a stable external contract that defines and governs the custom processes for publishing and retrieving data. Through these processes, you interact indirectly with the ClaimCenter database.

The technical realization of this contract starts with two mandatory components and an optional component. The mandatory components include a mapping and a schema. The optional component is a filter.

The mapping is a set of declarations that transform an object of a particular type first into an intermediate object that conforms to a specified schema. In the Integration Views context, the ultimate serialized form of this intermediate object is a JSON object or XML element.

The schema defines the structure of the data involved in the custom processes. The schema also enables you to use off-the-shelf tools to generate objects and methods to work with the data once it is serialized.

Whereas a mapping implements the transformation of an object of a particular type, the schema is like an interface that defines the structure of the transformed object.

The optional filter specifies the parts of the schema to use as output or for presentation. For example, suppose you wish to print out a legal document that represents parts of a transformed policy object. Suppose these parts consist of the effective dates during which a policy is active. The filter enables you to isolate this information from the transformed policy object for printing. In the Integration Views context, you specify a filter using GraphQL syntax.

The following table recapitulates the component parts of an Integration View:

Component	Description
Mapping	Instructions that convert an object of a particular type into an object that satisfies a specified schema
Schema	Desired structure of a transformed data object.
Filter	Isolation of the schema parts matching the parts of a transformed data object to output or present

The end result of an Integration View is a JSON object or XML element. To get to this point, you provide an input object of a particular type as a parameter to a mapping. This mapping transforms the input object into an intermediate object. The intermediate object has a structure that matches a particular schema. An optional filter isolates or selects the parts of the schema that match parts of the object. This process results in the intermediate object comprising transformed and isolated object parts. A JSON object or XML element containing a serialized version of these results serves as output.

Note:

An external handler calls the various Integration Views methods. These method calls transform input to output for a set of one or more objects. Types of handlers include event message handlers or REST API handlers.

Creating an Integration View

Creating an Integration View involves developing three components—a mapping file, a schema, and a filter. The mapping file and schema are mandatory. The filter is optional. The mapping file and schema are in JSON format. The filter is in GraphQL format.

Note: A set of one or more handler methods are necessary to support an Integration View but are not a part of the Integration View.

You can also convert existing GX models to Integration Views. The base configuration of ClaimCenter includes a Gosu class, `gw.json.gxconvert.GXModelConverter`, that you can use to convert the XML of the GX model to JSON format.

See also

- “Converting GX models to Integration Views” on page 650
- “Handling an Integration View using a REST API” on page 635

Mapping a data object to an Integration View schema

Mappings are declarative means for transforming a particular root object into a JSON object. The JSON object in turn conforms to a specified schema.

A mapping file contains the set of mappings. Each mapping in the mapping file corresponds to a mapper. Each of the mappings also corresponds to a particular type in the schema.

While the mappings relate to a schema, the file containing the mappings, the mapping file, is separate from the schema. This physical separation allows the mapping file and the schema to evolve independently. Consider the mappings to be an implementation and the schema a design. As such, best practice dictates decoupling the two.

The mapping file must contain the following information:

- Name and version of the schema that is subject to the mapping
- Type mappers, each of which corresponds to a particular type in the schema

The type mappers in the mapping file must contain the following information:

- Name of the schema definition for which they are producing output
- Specification of a root object for transforms
- Specification of property mappers for each object property in the target schema with a relative path from the root object

The following example contains an initial version of a mapping for contacts. The mapping corresponds to an initial version of a schema for such contacts. The mapping includes two type mappers for `Contact` and `Address` objects. These type mappers cover collectively three properties for a contact name, contact addresses, and the first line of an address:

```
{  
  "schemaName" : "gw.pl.contact-1.0",  
  "mappers": {  
    "Contact" : {  
      "schemaDefinition" : "Contact",  
      "root" : "entity.Contact",  
      "properties" : {  
        "Name" : {  
          "path" : "Contact.Name"  
        },  
        "Addresses" : {  
          "path" : "Contact.Addresses",  
          "mapper" : "#/mappers/Address"  
        }  
      }  
    },  
    "Address" : {  
      "schemaDefinition" : "Address",  
      "root" : "entity.Address",  
      "properties" : {  
        "AddressLine1" : {  
          "path" : "Address.AddressLine1"  
        }  
      }  
    }  
  }  
}
```

Note that the name of the JSON schema in the `schemaName` variable and in Guidewire Studio must be fully-qualified. In addition, each mapper has a `schemaDefinition` variable that refers to the name of an object or variable in the schema. Lastly, note that you can use a Gosu expression to specify the relative path from the root object for a property.

See also

- “Accessing and naming files related to Integration Views in Guidewire Studio” on page 635

Creating a schema for Integration View objects

An Integration View requires a schema to enable a mapping to transform JSON objects. You define a schema using a Schema Definition Language or SDL. Guidewire has adopted a subset of JSON Schema as the SDL for Integration Views.

You can import or combine a schema into another schema or a mapping file by using the schema name and version. The schema name and version come from the schema file name.

The code example to follow is a schema for `Contact` objects. The code example starts with a reference to the applicable SDL version. The remainder of the code sample defines `Contact` and `Address` objects. Moreover, the remainder defines the `Name` and `Addresses` properties for `Contact` objects and the `AddressLine1` property for `Address` objects:

```
{  
  "$schema": "http://json-schema.org/draft-04/schema#",  
  "definitions": {  
    "Contact" : {  
      "properties" : {  
        "Name" : {  
          "type" : "string"  
        },  
        "Addresses" : {  
          "type" : "array",  
          "items" : {  
            "$ref" : "#/definitions/Address"  
          }  
        }  
      }  
    },  
    "Address" : {  
      "properties" : {  
        "AddressLine1" : {  
          "type" : "string"  
        }  
      }  
    }  
  }  
}
```

```
    }
}
```

Note: The term *JSON Schema* refers to a Schema Definition Language or SDL, a subset of which Guidewire has adopted for Integration Views. The term *JSON schema* refers to a specific schema or schema file written in the JSON Schema SDL.

- “Accessing and naming files related to Integration Views in Guidewire Studio” on page 635
- <http://swagger.io/specification/>
- <https://spacetelescope.github.io/understanding-json-schema/structuring.html>
- <http://json-schema.org/>

Importing and creating an alias for another schema

Guidewire InsurancePlatform Integration Views permits a first JSON schema to import and create an alias for a second JSON schema. The second JSON schema must be defined in the same configuration environment as the first JSON schema. Importing the second schema allows the first schema to reference the second schema in place. This is to be distinguished from combining a second schema, which brings the second schema into the same namespace as the first schema.

The syntax for the import reference for JSON schemas is as follows:

```
"x-gw-import" : {
  "<alias>" : "<schemaName>-<versionNumber>",
},
```

In this syntax, the *alias* variable contains the name by which the first schema will refer to the second schema. The *schemaName* variable contains the fully-qualified name of the second schema. The *versionNumber* variable contains the full version number of the second schema.

The following code example supplies values for each of the variables in the syntax. The example provides an alias, *address*, for version 1.0 of the second schema, *gw.pl.address*:

```
"x-gw-import" : {
  "address" : "gw.pl.address-1.0",
},
```

See also

- “JSON schema file specification” on page 836

Additional characteristics of Integration View schemas

The following are notable additional characteristics of Guidewire Integration View schemas:

- All types are top-level. A type can reference another type but cannot contain one. Integration View schemas do not permit nested compound types.
- Scalar properties use built-in JSON Schema types, except for `null`. The allowed built-in JSON Schema types include `string`, `number`, `boolean`, `object`, `array`, and `integer`.
- The `format` attribute is for indicating generally recognized and custom formats. Among other examples, permitted values for the `format` attribute include `int32`, `int64`, `gw-biginteger`, and `gw-money`.
- Guidewire Integration View schemas do not permit enumeration constraints for typekeys. You can convert typekeys into enumerations for external consumers by setting `x-gw-export-enumeration` to `true`.

See also

- “JSON schema file specification” on page 836
- “JSON data types and formats” on page 847

Filtering Integration View objects for select attributes

Particular use cases of the Integration Views feature will require only a subset of the data from an Integration View object. You might want to exclude unnecessary data both from the data retrieval process and from the data serialization process. Excluding unnecessary data in these cases would reduce database load and payload size. One example use case where the Integration Views feature could fulfill this purpose is application-to-application integrations.

An Integration View generally defines the full set of data that a client can obtain. An optional filter for an Integration View will whitelist a subset of this data to optimize performance. Without the filter, the intermediate object and the serialized form of the object that the Integration View produces will contain the full set of data. With the filter, the output will contain the whitelisted subset of this data.

You define a filter using a subset of the GraphQL syntax. You save the filter to a file having a .gql extension. The name of the file follows the same pattern as schema files and mapping files: <package>.<name>-<version>.gql.

For example, to select only names from Integration View Contact objects, you would place the following code in a file called gw.p1.contact_names-1.0.gql:

```
{Name}
```

See also

- “Accessing and naming files related to Integration Views in Guidewire Studio” on page 635

Accessing and naming files related to Integration Views in Guidewire Studio

Access and name the various types of files related to Integration View files in Guidewire Studio by using the file locations and name formats in the following table:

Related file type	File location	Name format
Mapping files	Subdirectories of config/integration/mappings	<name>-<version>.mapping.json
JSON schemas	Subdirectories of config/integration/schemas	<name>-<version>.schema.json
Filters	Subdirectories of config/integration/filters	<name>-<version>.gql
Handlers	Any Guidewire Studio package	<name>.gs

Note: Handlers are listed in this table because they are necessary to use Integration Views. However, handlers are not a part of Integration Views.

Handling an Integration View

A set of one or more handler methods are necessary to support an Integration View. The handler methods call a set of other methods to query, select, map, filter, and transform data.

Handler methods are not a part of the Integration Views feature but are required to support an Integration View. They include REST API handler methods and event message handler methods.

Handler methods often take an input to identify data objects and return an output in the form of a set of transformed JSON objects. The bodies of these methods interact with a database to publish or retrieve data. You insert the resulting handler methods into a set of one or more standard Gosu classes.

Handling an Integration View using a REST API

The following two code examples contain REST API handler methods to obtain Contact data objects from a database. The first method returns a transformed JSON object with all of the information available about a single contact in the ClaimCenter database.

The first REST API handler method queries a single Contact data object by a public ID and obtains a mapper for such an object. This mapper is based upon a schema defining the resulting JSON object. The method then uses the

corresponding schema and transforms the Contact data object into a JSON object. Lastly, the method returns the JSON object:

```
function getContactByPublicId(publicId : String) : TransformResult {
    var query = Query.make(Contact).compare(Contact#PublicID, Relop.Equals, publicId)
    var result = query.select().FirstResult
    var mapper = JsonConfigAccess.getMapper("gw.pl.contact-1.0", "Contact")
    return mapper.transformObject(result)
}
```

The second REST API handler method returns a set of transformed JSON objects with only the name of each contact in the ClaimCenter database. The second method is similar to the first method with three exceptions. First, the second method obtains a set of transformed JSON objects, not necessarily one JSON object. Second, the second method requires the short-form name of a filter as an argument. Third, the second method constructs a `JsonMappingOptions` object to apply the `names` filter if this filter is the argument:

```
function getContacts(filter : String) : List<TransformResult> {
    var query = Query.make(Contact)
    var resultSet = query.select()
    var mapper = JsonConfigAccess.getMapper("gw.pl.contact-1.0", "Contact")
    var mappingOptions = new JsonMappingOptions()
    if (filter == "names") {
        mappingOptions.withFilter("gw.pl.contact_names-1.0")
    }
    return mapper.transformObjects(resultSet, mappingOptions)
}
```

See also

- “Accessing and naming files related to Integration Views in Guidewire Studio” on page 635

Handling an Integration View using an event message handler

The following six code blocks show how to use Integration Views with event message handlers. The examples collectively implement the following steps:

1. Obtain a `JsonMapping` object for a particular mapping file.
2. Obtain a `JsonMapper` object from the `JsonMapping` object.
3. Invoke the `transformObject` method or the `transformObjects` method on the `JsonMapping` object to produce respectively a `TransformResult` object or a list of `TransformResult` objects.
4. Serialize the `TransformResult` object into the JSON or XML language.

To obtain a `JsonMapping` object, use the `getMapper` method from the `JsonConfigAccess` class:

```
var mapping = JsonConfigAccess.getMapper("gw.pl.contact-1.0")
```

Once you have the `JsonMapping` object, retrieve a mapper by invoking the `Mappers` property `get` method. Use as an argument the type of the object for which you seek a mapper. In the following code example, the object type is `Contact`:

```
var mapper = mapping.Mappers.get("Contact")
```

You can collapse the previous two method calls into one call. Effect this collapse by using a helper method on the `JsonConfigAccess` class. The name of this helper method is `getMapper`. Invoke the `getMapper` method with the names of the mapping file and mapper as arguments. In the following code example, `gw.pl.contact-1.0` is the name of the mapping file and `Contact` is the name of the mapper:

```
var mapper = JsonConfigAccess.getMapper("gw.pl.contact-1.0", "Contact")
```

Once you obtain a `JsonMapper` object, define the object to be transformed. Then, invoke either the `transformObject` method or the `transformObjects` method on the `JsonMapper` object. Use the object to be transformed as an argument.

Depending on whether you call the `transformObject` method or the `transformObjects` method, the respective outcome of the invocation will be a `TransformResult` object or a `List` of `TransformResult` objects:

```
var contact : Contact // You usually obtain the Contact object from the event message rule context or from a query.  
var transformResult = mapper.transformObject(contact)
```

You can apply an optional filter to the mapping by adding an argument to the `transformObject` or `transformObjects` method. This additional argument would be a `JsonMappingOptions` object. On the `JsonMappingOptions` object, you can call the `withFilter` method with an argument containing the name of a GraphQL filter:

```
var transformResult = mapper.transformObject(contact, new  
JsonMappingOptions().withFilter("gw.pl.contact.contact_summary-1.0"))
```

After producing a `TransformResult` object or a `List` of such objects, serialize it into JSON or XML language. To serialize the `TransformResult` object into JSON language, use the `toJsonString` method:

```
var jsonString = transformResult.toJsonString()
```

To serialize the `TransformResult` object into XML language, use the `toXmlString` method:

```
var xmlString = transformResult.toXmlString()
```

The `TransformResult` object exposes additional methods to allow traversing the tree of values in the object. You can use these methods to build custom serialization formats for the object data.

chapter 31

GX models

A GX model enables the properties of a data type to be converted to XML. Supported data types that can be converted include Gosu classes and business data entities, among others. A GX model can include all or a subset of an associated data type's properties. Use GX models to limit the transfer of object data to those properties you need or desire to conserve resources and improve performance.

A GX model is a data type that is associated with another data type. The model includes the desired properties of the associated type that will be subsequently converted to XML. A GX model can include all or a subset of the associated type's properties. For example, a GX model called `AddressModel1` can be associated with the `Address` entity. The definition of the model might include the `PublicID` and `Description` properties from `Address`.

Even though the GX model contains properties of another entity, the internal formats of the model's properties are undefined. The model's internal format can change in different ClaimCenter versions. Best practice is not to depend on the internal format of GX model properties. Accordingly, do not access or modify a GX model's properties. Supported GX model operations include defining and creating models and converting their contents to XML.

GX models are defined in the Studio GX model editor. Using the editor, the properties of a data type are added to the GX model. As the model is defined, the editor automatically generates an XSD schema that specifies the XML structure of the model. Because of this automation, the XML definitions that the editor generates will be valid and do not require further validation.

When configuration code subsequently creates an instance of the GX model, the constructor methods accept an argument of an instance of the data type associated with the model. The properties of the argument that are referenced by the GX model are stored in the new instance. They can be subsequently converted to XML that conforms to the model's XSD schema. Continuing with the `Address` and `AddressModel1` example, an instance of the `AddressModel1` is created. The model's constructor accepts an instance of `Address` as an argument. The `PublicID` and `Description` properties of the argument are stored in the created GX model object. The model properties are subsequently converted to XML using one of the supported conversion methods.

Note: GX models functionality is superseded by Guidewire InsurancePlatform Integration Views.

See also

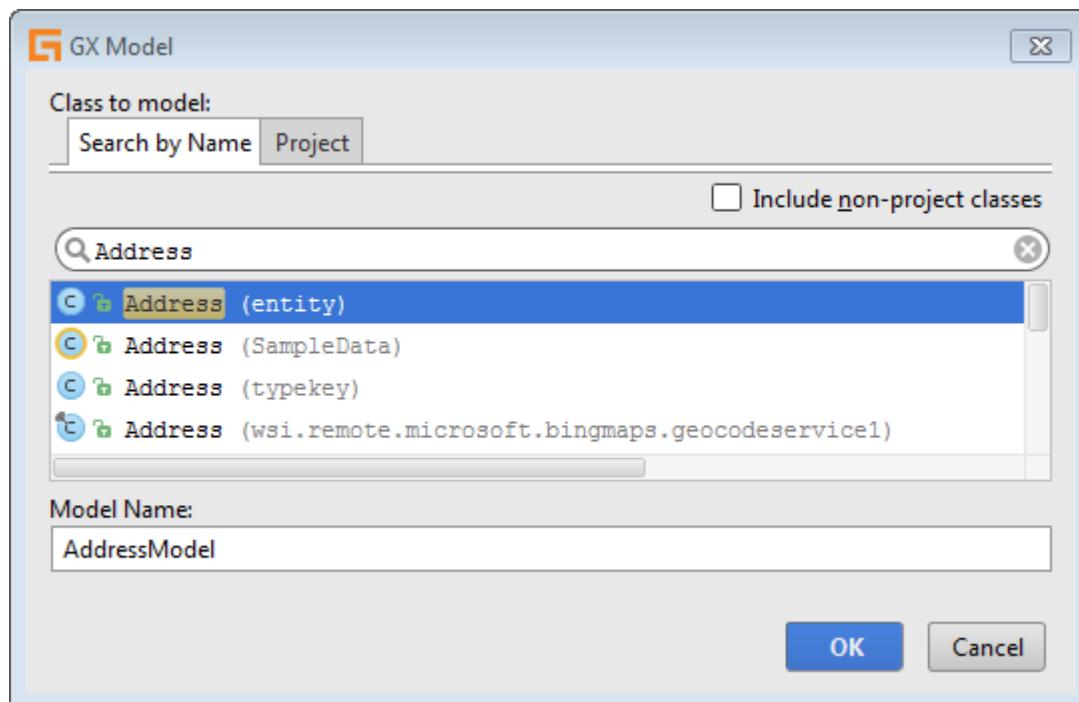
- “Guidewire InsurancePlatform Integration Views” on page 631

Create a GX model

Procedure

1. In Studio, navigate in the resource tree to the location **configuration > gsrc**.

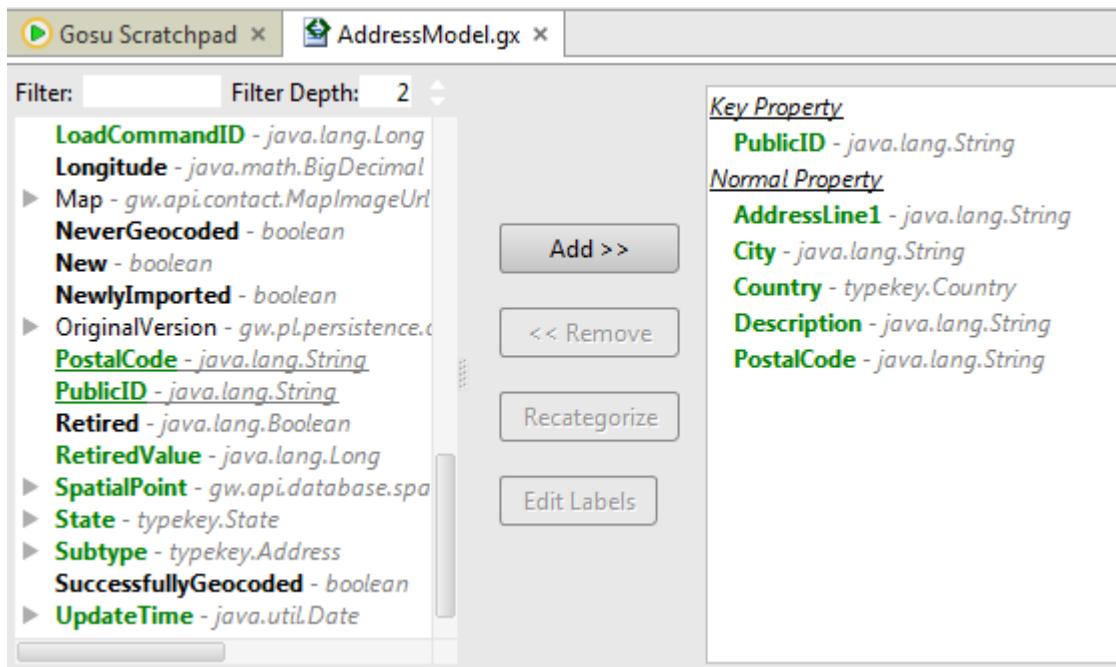
2. Within the class tree, navigate to the package in which you want to store your XML model, just like you would for creating a new Gosu class. If you need to create a top-level package, right-click on the folder icon for **gsrc**, and select **New Package**.
3. Right-click on the desired package. From the contextual menu, choose **New > GX Model** to open the **GX Model** window.
4. Select either the **Search by Name** or **Project** tab to list the available classes and data types to map.
5. In the Search field, enter the name of the type to export in XML format. For example, to map and export an **Address** entity, type **Address**. The dialog box lists the types that match any partial name you type. Select the desired class from the list.



6. Enter the desired GX model name in the **Model Name** field. The default name of the GX model is the name of the selected class followed by the word **Model1**. For example, if the selected class is **Address**, the model name is **AddressModel1**.

The GX model name defines the package hierarchy for the model, so choose its name carefully. For example, if an XML model called **AddressModel1** is created in the **mycompany** package, the fully qualified type for the **Address** entity in the GX model is **mycompany.addressmodel1.Address**.

7. Click **OK** to open the GX model editor.



8. To map a property to the GX model, select the property in the editor's left pane.

To navigate through a property's lower levels, click the arrow icon in front of the property to open the hierarchy below it. Continue navigating to lower levels until the desired property is located.

Alternatively, locate the desired property by entering its name in the **Filter** field. Studio searches for the property in lower levels to the depth specified in the **Filter Depth** field. For example, to find the myProp property in the hierarchy myClass.propertyOne.subProperty.myProp, set the **Filter Depth** to three to search three levels below the parent myClass.

9. With a property selected in the left pane, click **Add>>** to show the show the Mapping Type popup.
10. In the Mapping Type popup, select the property's mapping type, either **Key Property** or **Normal Property**. The selected property is added to the editor's right pane.
11. Continue adding properties to the GX model until the model is complete.

Results

The XML definition of the GX model can be viewed by selecting the **Text** tab at the bottom of the editor.

In the editor's lower pane, select the **Schema** tab to view the XSD schema that is automatically generated for the GX model. Select the pane's **Sample XML** tab to view a sample XML file that conforms to the XSD schema.

What to do next

For more information about key properties and normal properties in the context of the incremental mode, see "The Incremental Option" under "GXOptions" on page 643.

Including a GX model inside another GX model

A property in a GX model can reference other GX models if the models already exist for that type. For example, suppose there is a type called **Inner**, and you have already created a GX model for this type called **InnerMainModel**. Now suppose there is a type **Outer** that includes a property of **Inner**, and you want to use **InnerMainModel** to model that data for sending to an external system.

As you edit a new GX model for the type **Outer**, browse to a property on **Outer** that has the property type of **Inner**. Click **Add**, then click **Normal Property**. Studio opens a dialog box in which you select among multiple GX models that exist for type **Inner**. In this case, click **InnerMainModel**, then click **OK**.

Mappable and unmappable properties

The visual characteristics of a property listed in the left pane of the GX Model editor indicates whether the property can be mapped to a GX model.

A mappable property is shown in bold text. All properties of simple types, such as `String`, `Date`, numbers, typecodes, and enumerations, can be mapped. If a property is currently mapped to a GX model, it is shown underlined.

An unmappable property is shown in regular text. Unmappable properties include complex types, such as custom Gosu classes and business entities like `Claim`.

Studio supports all variants of properties, including the following types.

- Entity properties defined in the data model configuration files backed by database columns
- Entity virtual properties
- Public properties and variables in Gosu classes
- Public properties and variables in Java classes
- Public properties implemented in Gosu enhancements

Studio shows entity properties backed by database columns in green. It shows all other entity properties and variables in black.

Methods are not properties and, therefore, can never appear in the property mapping hierarchy. If you want the return result of a method to be a mapped property, add a Gosu enhancement property (a `property get` function) to return the desired value.

For entities such as `Claim` and `Address`, some properties are backed by database columns in the Guidewire data model. However, whether a column is backed by a database column does not affect whether that property contains a mappable type. Simple properties like numbers and `String` values are automatically mappable. Foreign key references to other entities are not mappable by default. However, you could map properties within such an other object entity, or you can create a GX model for the other object.

Normal and key properties

When a property is mapped to a GX model, it can be either a key property or a normal property.

A key property is typically used as a unique identifier for an object. For Guidewire business entities, the `PublicID` property is the property that identifies the entity to an external system. Accordingly, the `PublicID` property is commonly used as the key property for a GX model.

However, the key property of a GX model does not have to uniquely identify an object. For example, a type might have an enhancement property that calculates a value using a complicated formula. If providing the final value to an external system is desired, the property can be mapped as a key property.

All properties in a GX model that are not key properties are normal properties.

Your choice of what properties to map affects the XSD. Whether you define a property as normal or key affects any run time XML output but does not affect the XSD. However, the actual XML that Gosu generates for an object depends on some configuration settings at run time. For example, these settings define whether to send the entire object graph or only the properties that changed.

Automatic publishing of the generated XSD schema

When the ClaimCenter application is running, the application publishes the GX model's XSD schema. Other applications can import the schema from ClaimCenter and use it to validate related XML files.

ClaimCenter publishes the XSD schema at the following URL.

```
http://HOSTNAME:PORT/cc/ws
```

The web server displays a list of XSD schema files. Click on a file to view it. If your external system imports XSD schema files using an HTTP URL, save the URL to bookmark and retrieve the schema file as needed. If the external system needs the schema file on disk, copy and paste the file contents into an appropriate application to save it. You can also view the XSD schema in the Studio GX Model editor.

Create an instance of a GX model

An instance of a GX model can be created by using the `create` static method. Given the GX model file name `MODELNAME` and type name `TYPENAME`, create an instance of the GX model using the syntax shown below. The GX model property values are initialized with the values contained in the `object` argument.

```
var xmlInst = YOURPACKAGE.MODELNAME.TYPENAME.create(object)
```

As an example, assume a GX model of the Address entity was created and named `AddressModel1`. In addition, the GX model is stored in the package `com.mycompany`. Finally, an instance of the Address entity called `myAddress` contains valid address information. The following statement creates XML variables based on the GX model and initializes them with data from `myAddress`.

```
var xmlAddressData02 = com.mycompany.addressmodel.Address.create(myAddress)
```

GXOptions

The behavior of the GX model `create` method can be modified by specifying options as an argument of type `GXOptions`. The `GXOptions` argument is optional.

```
var xmlInst = YOURPACKAGE.MODELNAME.TYPENAME.create(object [, GXOptions])
```

The Incremental option

The `Incremental` option specifies whether to export to the GX model only properties that have changed on the base object. To determine whether a property has changed, ClaimCenter uses the current bundle to examine which entities were added, removed, or changed.

The `Incremental` option accepts a Boolean value. Its default value is `false`.

- `true` – Export only changed properties to the GX model.
- `false` – Export appropriate properties to the GX model, regardless of whether they have changed.

The following code statements demonstrate how to enable the `Incremental` option.

```
// Create a GX model with the Incremental option enabled
var myGxModel = com.mycompany.addressmodel.Address.create(myAddr,
    new gw.api.gx.GXOptions() {:Incremental=true})

// Alternatively, create and use a GXOptions variable with the Incremental option enabled
uses gw.api.gx.GXOptions
public static final var IncrementalOption : GXOptions = new GXOptions() {:Incremental=true}
var myGxModel02 = com.mycompany.addressmodel.Address.create(myAddr, IncrementalOption)
```

When the `Incremental` option is enabled, special behavior exists for key properties (as opposed to normal properties) on the root entity. Key properties on the root entity that have not changed are still exported to the GX model. The reason for this behavior is because key properties rarely change value. However, this behavior applies only to the root entity. If a key property in a subobject of the root entity has not changed, the property is not exported to the GX model.

Determining whether a property has changed is not always possible. Virtual properties and enhancement properties are not defined in the data configuration file and, as a result, they are not stored in the database. This condition prevents Gosu from determining whether a virtual or enhancement property has changed. In such situations, the setting of the `Incremental` option is ignored for the property, and the property is exported to the GX model.

Also, Gosu cannot determine whether a property in subobjects of virtual and enhancement properties has changed. Therefore, the `Incremental` option is ignored for the subobjects and all appropriate properties in the subobjects are

exported to the GX model. For example, the `PrimaryAddress` for a `Claim` entity is a virtual property. As a result, Gosu cannot determine whether the `PrimaryAddress` property and any property in the subobjects of `Claim.PrimaryAddress` have changed.

The SuppressExceptions option

The `SuppressExceptions` option specifies whether to suppress exceptions that occur when exporting properties to the GX model. An exception can occur when attempting to export a virtual or enhancement property.

The `SuppressExceptions` option accepts a Boolean value. Its default value is `false` so exceptions are not suppressed.

After the GX model instance has been created, the `$HasExceptions` property on the object indicates whether one or more exceptions occurred during its creation. Details of each exception can be retrieved by calling the `eachException` method, which expects a Gosu block. The block accepts a single argument that specifies an exception. Gosu calls the block for each exception.

The Verbose option

The `Verbose` option specifies whether to export a property to a GX model even if the property's value is `null`.

The `Verbose` option accepts a Boolean value. Its default value is `false` so a property with a value of `null` is not exported to the GX model.

A property with a value of `null` is exported to XML with the attribute `xsi:nil="true"`. An example property called `myNullProperty` is shown below.

```
<myNullProperty xsi:nil="true">
```

Properties that exist in subobjects to the `null` property are naturally also `null` values, but they are not exported to the GX model. Only the top-level `null` property is exported.

GX model labels

When an instance of a GX model is created, all the model's properties are included in the new object. Alternatively, GX model labels enable the object to include only a subset of the model's properties. A label is assigned to one or more of the model's properties. When an instance of the model is created and the label is specified, only the properties assigned the label are included in the new object.

A GX model can have multiple labels assigned to its properties, and a single property can have multiple labels assigned to it. When a GX model object is created, the appropriate combination of labels are specified to include the desired properties.

By using labels, a single GX model can be defined and each instance of the model can include different properties depending on the context. Imagine the following situation.

- External SystemA needs to include one key property and five standard properties on an entity.
- External SystemB needs to include two key properties and ten standard properties on the same entity.

Without labels, each system must create its own separate GX model. With labels, a single GX model can be created where the properties that interest SystemA are assigned a special label, such as `SysA`. In the same GX model, the SystemB properties are assigned a label like `SysB`. A property can have multiple labels assigned to it, which enables a single property to be used by both SystemA and SystemB. When SystemA converts the model properties to XML, only the properties with the `SysA` label are converted. Similarly, when SystemB converts the same model, only the properties with the `SysB` label are converted. The result is that a single GX model is used to handle the requirements of both SystemA and SystemB.

A GX model label applies to a parent entity and all child entities. For example, the `AddressModel` GX parent entity might include a child entity called `Country`. The `myGxLabel` can be assigned to properties in both the parent `AddressModel` and the child `Country`. When the GX model object is created and the `myGxLabel` is specified, the resulting object includes all properties in the parent and child that were assigned the label.

Assign a label to a GX model property

Procedure

1. In the right pane of the GX Model editor, select the property to receive the label.
2. Click the **Edit Labels** button to show the **Label Edit** popup.
3. Enter the desired label name in the text field. Names must conform to the requirements for Java identifiers. Labels must start with a letter, dollar sign (\$), or underscore (_). Subsequent characters can consist of any of those characters or a number.
4. Click the plus (+) button to assign the label to the property and add it to the list of property labels.
5. Assign additional labels as needed. When finished, select OK.

Reference a GX model label

You can reference a GX model label with either a static property or a String constant. A GX model label is a static property on the **Label** property of a model type. For example, to reference the **SysA** label on the GX **AddressModel** as a static property, use the following syntax:

```
AddressModel.Address.Label.SysA
```

If the GX **AddressModel** is located in the package **com.myCompany**, the complete reference to the **SysA** label is as follows:

```
com.mycompany.addressmodel.Address.Label.SysA
```

You can assign the **SysA** label reference to a variable with the following syntax:

```
var labelSysA = com.mycompany.addressmodel.Address.Label.SysA
```

A label name used by two unrelated models produces two separate and distinct labels. You reference these labels individually with static properties. In the following example, the **myLabel** name is used by both the **AddressModel** GX model and the unrelated **ContactModel** GX model:

```
var AddressLabel = com.mycompany.addressmodel.Address.Label.myLabel
var ContactLabel = com.mycompany.contactmodel.Address.Label.myLabel
```

Instead of referring to a GX model label by its static property, you can refer to the label by its name. To do so, use a **String** constant value. For the previous example, use "myLabel."

The differences between the **String** constant method and the static property method are a matter of convenience. You can use both to refer to the same GX model label. However, the **String** constant method is often more convenient than the static property method. As a case in point, the previous code example requires two static properties to refer to two separate and distinct labels having the same name. By contrast, you only need to use one **String** constant, "myLabel", to refer to both labels. While the static property referencing method can only refer to one individual GX model label at a time, the **String** constant referencing method can refer to multiple labels at once and requires fewer keystrokes.

Create an instance of a GX model with labels

An instance of a GX model with labels can be created using the **create** static method.

Use the **create** static method with the list of relevant label references specified as an argument. The list of label references is specified within curly braces using either the static property method or the **String** constant method. Multiple label references are separated by commas. If an argument specifying GX model options is included, it is located immediately before the list of labels. The syntax with labels referenced by static property is shown below.

```
var xmlInst = YOURPACKAGE.MODELNAME.TYPENAME.create(object [, GXOptions] [, { Label [, Label2] }])
```

Some examples are shown below.

```
// Create variables that reference the labels by their static properties
var labelSysA = com.mycompany.addressmodel.Address.Label.SysA
var labelSysB = com.mycompany.addressmodel.Address.Label.SysB
// Create instances of the GX model that include various labels referenced by their static properties
var modelSysA = com.mycompany.addressmodel.Address.create(myAddress, {labelSysA})
var modelSysAB = com.mycompany.addressmodel.Address.create(myAddress, {labelSysA, labelSysB})
```

If the labels are to be ignored and all of the properties in the model are to be included in the instance, refer to the reserved label `default_label` in the creation statement by using the static property method. You can also refer to this label implicitly by using the static property method with no arguments.

```
// Create variable that references the default_label
var defLabel = com.mycompany.addressmodel.Address.Label.default_label
// Include all properties in the model and ignore labels
var modelTotalA = com.mycompany.addressmodel.Address.create(myAddress, {defLabel})
```

Labels referenced using the `String` constant method are included in a separate list of labels in the model's `create` code statement. The list of `String` constant label references is located after the list of labels referenced by their static properties. Each list of label references is specified within curly braces with multiple label references separated by commas.

```
var xmlInst = YOURPACKAGE.MODELNAME.TYPENAME.create(object,
    [, GXOptions]
    [, { staticLabel [, staticLabel2] }]
    [, { strConstantLabel [, strConstantLabel2] }])
```

Note: `String` constant label references must match the label name for an existing GX model. Be sure to spell `String` constant label references properly and in the correct case. You can use a `String` constant to refer to the reserved label, "default_label", explicitly. You can also use the `create` static method with no arguments to refer to the same reserved label implicitly.

If the list of `String` constant label references is specified, then the preceding list of static property label references must also be specified. This requirement applies even if the preceding list is designated as an empty list. Both lists can be populated with items.

Some example code statements are shown below.

```
var label001 = com.mycompany.addressmodel.Address.create(myAddress, {}, {"myLabel001"})
var label002 = com.mycompany.addressmodel.Address.create(myAddress, {}, {"myLabel001", "myLabel002"})
var label003 = com.mycompany.addressmodel.Address.create(myAddress, {labelSysA}, {"myLabel001"})
var label004 = com.mycompany.addressmodel.Address.create(myAddress, {}, {"default_label"})
```

Static property label references and `String` constant label references vary in their dependence on the GX models to which they relate. On the one hand, static property label references are specific to the GX models on which they are defined. This is true even if the labels for the references have the same name. On the other hand, `String` constant label references apply across GX models or within multi-level GX models.

For example, suppose that a GX model `GxModel1` defines a property `Property1_1` with label `SystemA`. Assume also that a GX model `GxModel2` defines a property `Property2_1` with a label having the same name, `SystemA`. The respective static property references for the `SystemA` label—`GxModel1.Label.SystemA` and `GxModel2.Label.SystemA`—go in the first labels argument position. Alternatively, the `String` constant reference, "`SystemA`", goes in the second labels argument position. The references of the static property variety are separate and specific to their corresponding GX models. By contrast, the reference of the `String` constant variety is model-independent and applies to both `Property1_1` on `GxModel1` and `Property2_1` on `GxModel2`.

When any single GX model within a parent and child hierarchy defines labels, the label syntax must be used when creating an instance of the parent. The following situations can exist.

- Parent model does not have labels, child model has labels – The child model can be located at any level below the parent. The parent model will export all of its properties. The child model will export its properties that are assigned the referenced label.

- Parent model has labels, child model does not have labels – The parent model will export its properties that are assigned the referenced label. The child model will export all of its properties.
- Both the parent and child model have labels – Both the parent and child models will export their properties that are assigned the referenced labels. A label can be used in both the parent and child models.

```
// AddressModel includes child entities.  
// Various properties in AddressModel and its child entities are assigned myGxLabel.  
// The resulting embeddedModel includes all properties in the parent and children with the label.  
var embeddedModel = com.mycompany.addressmodel.Address.create(myAddress, {}, {"myGxLabel"})
```

GX model label example

For the example described in this section, assume that the `Address` entity was used to create a GX model called `AddressModel` in a package called `com.mycompany`. Several properties are included in the model, but only the `PublicID` and `AddressLine1` properties have the label `myAddressLabel` assigned to them.

```
// Create an Address GX model object  
var myAddress = new Address()  
myAddress.PublicID = "example1234"  
myAddress.AddressLine1 = "123 Main St."  
myAddress.City = "Foster City"  
myAddress.Description = "City on the Bay"  
myAddress.PostalCode = "94404"  
var addr = addressmodel.Address.create(myAddress, {addressLabel})  
  
// Get references to myAddressLabel and the default_label in the AddressModel GX model  
var addrLabel = com.mycompany.addressmodel.Address.Label.myAddressLabel  
var defLabel = com.mycompany.addressmodel.Address.Label.default_label  
  
// Create an instance of the GX model that includes only the properties with the label myAddressLabel  
var addrModelLabels = com.mycompany.addressmodel.Address.create(addr, {addrLabel})  
  
// Create an instance of the GX model that includes all properties, regardless of labels  
var addrModelAll = com.mycompany.addressmodel.Address.create(addr, {defLabel})  
  
// Output the XML contents of each model variable  
print(addrModelLabels.asUTFString())  
print(addrModelAll.asUTFString())
```

The output of the `addrModelLabels` variable is shown below. The output of the `addrModelAll` variable is similar, but includes all the properties contained in the model.

```
<?xml version="1.0"?>  
<Address xmlns="http://guidewire.com/bc/gx/com.mycompany.addressmodel">  
  <PublicID>example1234</PublicID>  
  <AddressLine1>123 Main St.</AddressLine1>  
</Address>
```

Serialize a GX model object to XML

Gosu provides multiple ways to serialize the contents of a GX model object.

- The `Bytes` property, which is the preferred technique
- The `asUTFString` method

When serializing the contents of a GX model object, validation of the model is disabled by default. The validation is unnecessary because it is not possible for the populated model to be invalid. Disabling the validation reduces the GX model heap space size, which can be significant for a bundle that contains hundreds of GX models.

Arrays of entities in XML output

Entities that were added or removed from the array employ special formatting in the XML. This feature only applies if the `Incremental` option is enabled.

In the entity array data model, the foreign keys are on the child entities pointing to the parent entity. Gosu hides this implementation detail and makes the child entities appear as a read-only array on the parent entity.

The following behaviors occur when the `Incremental` option is enabled.

- If an element in an entity array or Java collection, such as `java.util.ArrayList`, is new in the database transaction, the element has the `action` attribute with the `String` value “`ADD`.” Because the entity is new, the entity and its subobjects fully export. The output for this entity performs as if the `Incremental` option was disabled.
- If an element in an entity array is removed in the database transaction, its element has the `action` attribute with the `String` value “`REMOVE`.” For removed elements in arrays with the `Incremental` option enabled, only the key properties are exported. The entity’s normal properties are not exported.
- If an element in an entity array does not have the `action` attribute, then the element was neither added nor deleted, but instead was changed.

During serialization, the `action` attribute is populated only if the `Incremental` option is enabled.

The `action` attribute only exists on:

- Elements that appear in the model as an array or Java collection such as a `java.util.ArrayList`.
- The root element of every GX Model. This element has the `action` attribute so that other models can include this model as a child model within an array. For example, suppose there are models for type A and for type B. The GX model for type A could include a property that has the type `B[]` or `ArrayList`.

Sending a message only if data model fields changed

To use a GX model in messaging code, edit the Event Fired rules to generate a message payload that uses the GX model.

After you pass a type to a GX model’s `create` method, check the `$ShouldSend` property. The `$ShouldSend` property returns `true` if at least one mapped data model field changed in the local database bundle. If you want to send your message only if data model fields changed, use the `$ShouldSend` property to determine whether data model fields changed.

```
if (MessageContext.EventName == "AddressChanged") {
    var xml = mycompany.messaging.addressmodel.Address.create(MessageContext.Root as Address)

    if (xml.$ShouldSend) {
        var strContent = xml.asUTFString()
        var msg = MessageContext.createMessage(strContent)

        print("Message payload for debugging:")
        print(msg);
    }
}
```

When Gosu checks for mapped data model properties that changed, Gosu checks both normal and key properties. Because a key property uniquely identifies an object, typically that value never changes.

An important exception to this behavior occurs if any ancestor of a changed property changed its value, including to `null`. In this case, the change might not trigger `$ShouldSend` to be `true`. If the original value of the non-data-model field indirectly referenced data model fields, those fields do not count as properties that trigger `$ShouldSend` to be true. If the property that changed has a path from the root type that includes non-data-model fields, Gosu’s algorithm for checking for changes does not determine that this field changed. This behavior also occurs if the property value changed to `null`. ClaimCenter can tell whether a property changed only if all its ancestors are actual data model fields rather than enhancement properties or other virtual properties.

For example, suppose that a user action causes a change to the mapped field `A.B.C.D` property and also `A.B.C` became `null`. If the `B` property and the `C` property are both data model fields, Gosu detects this bundle change and sets `$ShouldSend` to `true`. However, if the `B.C` property is implemented as a Gosu enhancement or an internal Java method, then `$ShouldSend` is `false`. This difference is because the bundle of entities does not contain the old value of `A.B.C`. Because Gosu cannot determine whether the property changed, Gosu does not mark the change as one that sets the property `$ShouldSend` to the value `true`.

This special behavior with non-data-model properties subgraphs is similar to how the `Incremental` property in `GXOptions` is effectively disabled in virtual property subgraphs.

Note that if a property contains an entity array and its contents change, `$ShouldSend` is `true`. In other words, if an entity is added or removed from the array, `$ShouldSend` is `true`.

The behavior of `$ShouldSend` does not change based on the value of the `Verbose` or `Incremental` options.

Conversions from Gosu types to XSD types

The following table lists the type conversions that occur when using a GX model to export a type from Gosu to XML.

Gosu type	GX Model XSD type
<code>boolean</code>	<code>xsd:boolean</code>
<code>byte</code>	<code>xsd:byte</code>
<code>byte[]</code>	<code>xsd:base64Binary</code>
<code>char[]</code>	Unsupported mapping
<code>double</code>	<code>xsd:double</code>
<code>float</code>	<code>xsd:float</code>
<code>gw.api.database.spatial.SpatialPoint</code>	<code>xsd:string</code>
<code>gw.api.database.spatial.SpatialPolygon</code>	<code>xsd:string</code>
<code>gw.api.financials.CurrencyAmount</code>	<code>xsd:string</code>
<code>gw.pl.currency.MonetaryAmount</code>	<code>xsd:string</code>
<code>gw.xml.xsd.types.XSDDate</code>	<code>xsd:date</code>
<code>gw.xml.xsd.types.XSDDateTime</code>	<code>xsd:dateTime</code>
<code>gw.xml.xsd.types.XSDDuration</code>	<code>xsd:duration</code>
<code>gw.xml.xsd.types.XSDGDay</code>	<code>xsd:gDay</code>
<code>gw.xml.xsd.types.XSDGMonth</code>	<code>xsd:gMonth</code>
<code>gw.xml.xsd.types.XSDGMonthDay</code>	<code>xsd:gMonthDay</code>
<code>gw.xml.xsd.types.XSDGYear</code>	<code>xsd:gYear</code>
<code>gw.xml.xsd.types.XSDGYearMonth</code>	<code>xsd:gYearMonth</code>
<code>gw.xml.xsd.types.XSDTime</code>	<code>xsd:time</code>
<code>int</code>	<code>xsd:int</code>
<code>java.lang.Boolean</code>	<code>xsd:boolean</code>
<code>java.lang.Byte</code>	<code>xsd:byte</code>
<code>java.lang.Double</code>	<code>xsd:double</code>
<code>java.lang.Float</code>	<code>xsd:float</code>
<code>java.lang.Integer</code>	<code>xsd:int</code>
<code>java.lang.Long</code>	<code>xsd:long</code>
<code>java.lang.Short</code>	<code>xsd:short</code>
<code>java.lang.String</code>	<code>xsd:string</code>
<code>java.math.BigDecimal</code>	<code>xsd:decimal</code>

Gosu type	GX Model XSD type
java.math.BigInteger	xsd:integer
java.net.URI	xsd:anyURI
java.net.URL	xsd:anyURI
java.util.Calendar	xsd:dateTime
java.util.Date	xsd:dateTime
javax.xml.namespace.QName	xsd:QName
long	xsd:long
short	xsd:short
Any Guidewire typekey	xsd:string

Converting GX models to Integration Views

The base configuration of ClaimCenter includes a Gosu class, `gw.json.gxconvert.GXModelConverter`, that you can use to convert the XML of existing GX models to the JSON format of Integration Views. This class does not provide a complete conversion. You must edit the new Integration Views to complete the conversion. You can edit the class to increase the number of structures in your GX models that are converted without manual edits to the Integration Views. Make your changes in the `processNode(node : SimpleXmlNode)` method and the methods that it calls.

`gw.json.gxconvert.GXModelConverter` is designed to be used in the Gosu Scratchpad in Guidewire Studio. This class provides a static public method, `convertGxModelsToJsonSchema`, to convert your GX models. This method has multiple signatures. Each takes a `String` array that contains the GX models file names and a `String` that contains the schema version as arguments and returns an array of `File` objects. These `File` objects are the mapping and schema files for the Integration Views that are created in the `modules/config/integration` folder hierarchy in the ClaimCenter folder. If you provide multiple GX models names to the `GXModelConverter` method, the result is a single Integration View that includes all the GX models. The name of the Integration View is the name of the first GX model or the name that you provide, with a suffix of the form `-schemaVersion` and an extension of `.json`.

The following signatures of the `convertGxModelsToJsonSchema` method are available:

`convertGxModelsToJsonSchema(fileNames : String[], schemaVersion : String) : File[]`

The minimal version of the method.

fileNames

An array of `String` values that are the paths of the GX models to convert. The paths are relative to the `configuration/gsrc` folder.

schemaVersion

A `String` value that is the numeric version of the mapping and schema files that the conversion generates.

`convertGxModelsToJsonSchema(fileNames : String[], genFileName : String, schemaVersion : String) : File[]`

This version of the method names the output files as specified by the `genFileName` parameter.

fileNames

An array of `String` values that are the paths of the GX models to convert. The paths are relative to the `configuration/gsrc` folder.

genFileName

The relative path and file name of the mapping and schema files in the `mapping` and `schema` folders in the `config/integration` folder hierarchy.

schemaVersion

A String value that is the numeric version of the mapping and schema files that the conversion generates.

convertGxModelsToJsonSchema(rootDirName : String, fileNames : String[], genFileName : String, schemaVersion : String) : File[]

This version of the method specifies the root folder that contains the GX models rather than the .

rootDirName

The folder that contains the GX models. You can provide a path that is relative to the ClaimCenter folder or an absolute path.

fileNames

An array of String values that are the paths of the GX models to convert. The paths are relative to the rootDirName folder.

genFileName

The relative path and file name of the mapping and schema files in the mapping and schema folders in the config/integration folder hierarchy.

schemaVersion

A String value that is the numeric version of the mapping and schema files that the conversion generates.

Example

The following Gosu code shows how to call gw.json.gxconvert.GXModelConverter for a GX model named MyAddressmodel.gx in the configuration/gsrc/doc/example folder:

```
uses gw.json.gxconvert.GXModelConverter

var myFiles = GXModelConverter.convertGxModelsToJsonSchema({"doc/example/MyAddressmodel.gx"}, "1.0")
print("2 arguments")
for (f in myFiles) {
    print("\t" + f.AbsolutePath)
}
myFiles = GXModelConverter.convertGxModelsToJsonSchema({"doc/example/MyAddressmodel.gx"}, "doc/newexample/myNewAddressIV", "1.0")
print("3 arguments")
for (f in myFiles) {
    print("\t" + f.AbsolutePath)
}
myFiles = GXModelConverter.convertGxModelsToJsonSchema("C:/myPolicyCenter-repository/modules/configuration/gsrc", {"doc/example/MyAddressmodel.gx"}, "doc/newiv/myNewAddressIV", "1.0")
print("4 arguments")
for (f in myFiles) {
    print("\t" + f.AbsolutePath)
}
```

This code produces the following output:

```
2 arguments
    C:\Guidewire\PolicyCenter\modules\configuration\config\integration\schemas\doc\example
    \MyAddressmodel-1.0.schema.json
    C:\Guidewire\PolicyCenter\modules\configuration\config\integration\mappings\doc\example
    \MyAddressmodel-1.0.mapping.json
3 arguments
    C:\Guidewire\PolicyCenter\modules\configuration\config\integration\schemas\doc\newexample
    \myNewAddressIV-1.0.schema.json
    C:\Guidewire\PolicyCenter\modules\configuration\config\integration\mappings\doc\newexample
    \myNewAddressIV-1.0.mapping.json
4 arguments
    C:\Guidewire\PolicyCenter\modules\configuration\config\integration\schemas\doc\newiv
    \myNewAddressIV-1.0.schema.json
    C:\Guidewire\PolicyCenter\modules\configuration\config\integration\mappings\doc\newiv
    \myNewAddressIV-1.0.mapping.json
```


Archiving integration

ClaimCenter supports archiving a claim as a serialized stream of data. Each serialized stream is a XML document. If your installation uses archiving, you must implement the `IArchiveSource` plugin interface. The plugin is responsible for storage and retrieval of archival data in the archive backing store. The archive backing store typically is on a different physical computer. The archive backing store can be anything that you choose. Common choices for an archive backing store include:

- Files on a remote file system
- Documents in a Document management system
- Large binary objects in the rows of a database table

ClaimCenter includes a demonstration implementation of the `IArchiveSource` plugin interface.

IMPORTANT:

Guidewire provides plugin implementation class `IArchiveSource` for demonstration purposes only. This implementation writes archival data as files to the local file system on the server. Do not use this class in a production system. Implement your own archiving plugin that stores data on an external system.

The following table summarizes the role of this plugin interface.

Plugin interface	Description	Implementation class
<code>IArchiveSource</code>	Provide methods to store and retrieve a serialized XML document that represents one instance of the <code>Claim</code> archive graph.	<code>ClaimInfoArchiveSource</code>

See also

- *Application Guide*

Overview of archiving integration

Basic integration flow for archiving storage

The following sequence describes the high-level integration flow for archival storage.

1. The BillingCenter Archive batch process runs.
2. ClaimCenter determines whether to skip or exclude claims from archiving.

To determine what claims to skip or exclude, ClaimCenter runs the archiving rule set. ClaimCenter uses the following entities and properties to determine archive eligibility:

- `ClaimInfo.ExcludedFromArchive` - Boolean flag that if set to `true` prevents ClaimCenter from even attempting to archive the claim. If the value is `true`, ClaimCenter skips the claim and does not run the archiving rule set for this claim again until this flag is reset to `false`. For an excluded claim, the `ClaimInfo.ExcludedReason` property contains a `String` that describes the reason for the exclusion.
 - `Claim.DateEligibleForArchive` - Date that determines the earliest archive eligibility. If the current date is before this date, ClaimCenter does not attempt to archive the claim nor does it run the archiving rule set for this claim.
3. ClaimCenter encodes the claim into an in-memory XML representation for one XML document.
 4. ClaimCenter serializes each XML document.
 5. ClaimCenter calls the archive source plugin that you created to store the data.
 6. The archive source plugin sends archival data to an external system.

IMPORTANT: Your implementation of this plugin interface must connect to your own archive backing store.

7. ClaimCenter deletes the claim and all its subtypes but preserves the root `ClaimInfo` entity. This stub entity represents the original claim but contains only high-level information for the claim.

Basic integration flow for archiving retrieval

The following sequence describes the high-level integration flow for archiving retrieval.

1. The user identifies a claim to retrieve.
2. Synchronously, ClaimCenter finds the associated `ClaimInfo` entity for the claim.
3. ClaimCenter tells the archive source plugin to retrieve the XML formatted archival data from the archive backing store.
4. ClaimCenter deserializes the XML data into an in-memory representation of the stored XML.
5. If the archival data is from an earlier data model of ClaimCenter, ClaimCenter next runs upgrade steps on the temporary object.

This upgrade activity includes running any built-in upgrade triggers as well as customer-written upgrade steps that extend built-in upgrade steps. For example, the customer upgrade steps might upgrade data in data model extension properties. For more information, see “Upgrading the data model of retrieved data” on page 654

6. ClaimCenter converts the temporary in-memory object into a real entity instance, upgrades the data model if necessary, and finally persists the entity instance to the database.

Upgrading the data model of retrieved data

During a ClaimCenter archive retrieval operation, after the archive source plugin retrieves archival data, ClaimCenter deserializes the data into an in-memory object that represents the entity instance. This in-memory object is not yet a real entity instance. ClaimCenter loads the data into an object called an archived object, represented by the interface `IArchivedEntity`.

ClaimCenter then determines if the archive object is from an data model version that is earlier than the current version. If this is the case, ClaimCenter runs upgrade steps on the object. This includes running any built-in upgrade triggers as well as running customer-written upgrade steps that extend the built-in upgrade steps, for example:

- ClaimCenter performs built-in upgrade steps such as adding a column to the object.
- ClaimCenter runs customer upgrade steps such as upgrading the data in the data model extension properties.

Only after all upgrade triggers run does ClaimCenter attempt to convert the data object into a real entity instance that matches the current data model of the application. If this process fails, the entire restore operation fails. If it succeeds, the application commits the new data to the application database.

ClaimCenter also links the retrieved data with the `ClaimInfo` entity.

See also

- “Customizing the database upgrade” in the *ClaimCenter Upgrade Guide*

Contact info entities and other archiving entities that persist

Most archiving documentation focuses on the main graph of entity types and the one `RootInfo` entity at the top of the graph. For ClaimCenter, the `RootInfo` entity is the `ClaimInfo` entity.

However, there are other `Info` entities, most notably the `ContactInfo` entity. The `ContactInfo` entity represents an archived contact. ClaimCenter persists the `ContactInfo` entity instance so that users can search for a claim by important contacts on the claim, even for archived claims. ClaimCenter creates `ContactInfo` entities only on archive. ClaimCenter creates `ContactInfo` entities for the following:

- The one contact representing the insured
- One or more contacts represent the claimants

Internal code implements the creation of these items. You can add more `ContactInfo` entities or custom entity types. To add additional `Info` entities, create the entities in the `updateInfoOnStore` and `updateInfoOnRetrieve` methods in the archive source plugin.

See also

- For information on the archive source plugin, see “Archive source plugin” on page 658.

Error handling during archive

If the archiving process fails in any way, consult both of the following:

- Application logs
- The Server Tools **Archive Info** screen

To view the **Archive Info** screen, you must have administrative privileges.

Archiving storage integration detailed flow

ClaimCenter implements archive integration as a work queue. ClaimCenter performs the following archiving tasks to create archive writers and workers for the work queue.

1. First, the application confirms that the archive backing store system is available. It is possible that the archive backing store is unavailable due to network problems or configuration issues. If the archive backing store is unavailable for any reason, then the writer work process logs an information message but does not create archive work items.
2. If the archive backing store is available, Archiving Item Writer batch processing finds entity instances that are potentially eligible for archiving. The batch process creates a work item as a row in the database for every entity instance that is eligible for archiving.

In ClaimCenter, the only eligible entity type for archiving is a claim.

The archive worker process performs all the following steps to process the archive items.

3. Each worker process performs eligibility checks on each work item. Guidewire defines some eligibility criteria in internal code.

For example, a claim is ineligible if it is a payee on a bulk invoice.

Additionally, ClaimCenter calls the rule set Default Group Claim Archiving Rules. That rule set contains additional eligibility checks. You can enable, disable, delete, or modify eligibility checks to meet your business requirements.

4. For each work item, ClaimCenter performs the following steps within a single database transaction:
 - a. The worker copies some internal properties from the `Claim` to the stub `ClaimInfo` entity instance.
 - b. The worker compares all data model extension properties on `Claim` and `ClaimInfo` entities. If any extension properties have the same name and compatible types, then the worker copies those property values from the `Claim` to the `ClaimInfo`.
 - c. The worker calls the `IArchiveSource` plugin method `updateInfoOnStore`. It takes a single argument, which is a `ClaimInfo` entity instance. In the plugin definition, this parameter is declared as an entity type that implements the `RootInfo` interface. A `ClaimInfo` entity instance encapsulates information about a single claim including its archive-related status information. If you need to add or modify properties on `ClaimInfo` entity instance in addition to properties mentioned in “step b”, set these properties in your `updateInfoOnStore` method.
 - d. The worker *tags* the archived root entity instance (the claim) specified in the work item. Next, the worker recursively tags all entity instances in the domain graph whose parent entity was tagged.
The process of tagging includes setting the `ArchivePartition` property on the root entity instance to a non-null value. The tagging process helps the application determine what data to delete at the end of the archiving process.
 - e. The worker internally generates a structure that represents an XML document for the archived `Claim` and its linked entities.
 - f. The worker serializes the XML structure into a stream of XML data as bytes. The worker outputs these bytes as an `java.io.InputStream` object.
 - g. The worker deletes most of the archived data. However, ClaimCenter does not delete the `ClaimInfo` entity instance. This entity instance remains in the database. It summarizes the archived claim including its archiving status.
 - h. The worker calls the archive source plugin `store` method. The first argument to the method is the input stream that contains the serialized XML document. The second argument is the `ClaimInfo` entity instance that encapsulates archive-related properties of the claim. In an actual production system, the archive source plugin sends the data to an external system along with any related metadata from the `ClaimInfo` entity instance. For example, the archive source might contain details that link the archival data to that `ClaimInfo` entity instance.
5. The worker commits the database transaction. This ends the database transaction for the preceding database changes. If any changes before this step threw an exception, all changes in this entire transaction are rolled back.
6. As the last step in the archiving process for each work item, the worker calls the archive source plugin method `storeFinally`. ClaimCenter calls this method independent of whether the archive transaction succeeded. For example, if some code threw an exception and the archive transaction never committed, ClaimCenter still calls this method. Use this method to do any final clean up. To make any changes to entity data, you must run your Gosu code within a call to `Transaction.RunWithNewBundle(block)`. That API sets up a bundle for your changes, and then commits the changes to the database in one transaction. For details, see the *Gosu Reference Guide*.

See also

- “Archiving claims from external systems” on page 154
- “Archive source plugin storage methods” on page 659
- “Archive source plugin utility methods” on page 662
- *Administration Guide*

Archiving retrieval integration detailed flow

As its name indicates, the archive *retrieval* operation fetches archived data from the archive backing store and inserts it into the application database. It is up to you to provide the necessary hooks into the retrieval process. The purpose of retrieval is to overwrite the data in the database with archived data. ClaimCenter does not check whether the database has a newer version of the record to retrieve from the archive.

For example, in ClaimCenter, with archiving enabled, the **Advanced Search** screen displays a **Retrieve from Archive** button if the search query returns an archived item.

Within ClaimCenter, the archive document retrieval process has the following overall flow:

1. The user initiates the retrieval process on an archived claim.
2. Synchronously, the retrieve request work queue finds items to retrieve.
3. ClaimCenter calls the `IArchiveSource` plugin method called `retrieve`. ClaimCenter passes this method a reference to a `RootInfo` entity instance. Your plugin implementation is responsible for returning the object binary data that describes the XML data that was originally stored. Your plugin implementation must provide this data as an `InputStream` object.

Note: ClaimCenter turns off field validation during the retrieval of archival data from the archive backing store.

4. ClaimCenter clears the data on `ClaimInfo` that it copied from the `Claim` entity instance during the initial archiving process.
5. ClaimCenter creates any data that relates to the retrieve. For example, notes and history events.
6. ClaimCenter performs any upgrade steps defined for the data.
See “Upgrading the data model of retrieved data” on page 654.
7. ClaimCenter calls `IArchiveSource.updateInfoOnRetrieve`. This method gives you a chance to perform additional operations in the same bundle as the retrieval operation, after ClaimCenter recreates all the entities but before the commit.

In the demonstration plugin implementation, this method sets the `Claim.DateEligibleForArchive` property to a date based on the `DaysRetrievedBeforeArchive` configuration parameter. This ensures that the application does not immediately attempt to archive the claim again but instead waits until it is eligible for archive.

8. ClaimCenter commits the bundle.
9. ClaimCenter calls the archive source plugin `retrieveFinally` method. It is up to you to decide what additional post-retrieval operations to implement. For example, use this method to delete the archive storage file or mark its status.

WARNING: Do not attempt to modify the retrieved data in the `retrieveFinally` method. Guidewire does not recommend, nor does Guidewire support, the use of this method to modify archive data after you retrieve the data.

Before you delete the returned XML document, first consider whether it is important to compare what the archived item looked like initially with a subsequent archive of the same item.

If the retrieval process does not complete successfully, then consult the application logs as well as the **Server Tools Archive Info** screen. You must have administrative privileges to access this screen.

See also

- “Archiving claims from external systems” on page 154
- “Archive source plugin retrieval methods” on page 661
- “Archive source plugin utility methods” on page 662
- *Application Guide*

- *Configuration Guide*
- *Administration Guide*

Archive source plugin

The use of archiving in Guidewire ClaimCenter requires the implementation of the `IArchiveSource` plugin interface. The plugin is responsible for storage and retrieval of archival data in the archive backing store. The archive backing store typically is on a different physical computer. The choice for the archive backing store can be:

- Files on a remote file system
- Documents in a document management system
- Large binary objects in the rows of a database table

ClaimCenter includes a demonstration implementation of the `IArchiveSource` plugin interface:

```
gw.plugin.archiving.ClaimInfoArchiveSource
```

The superclass of the plugin implementation is the base class `gw.plugin.archiving.ArchiveSource`.

IMPORTANT: Guidewire provides plugin implementation class `ArchiveSource` for demonstration purposes only. This implementation writes archival data as files to the local file system on the ClaimCenter server. Do not use this class in a production system. Implement your own archiving plugin that stores data on an external system.

Storing archival data

To store archival data, ClaimCenter calls `ArchiveSource` method `store` synchronously, waiting for the method to complete or fail.

Retrieving archival data

To retrieve archival data, ClaimCenter calls `ArchiveSource` method `retrieve`. The single argument to the `retrieve` method is an entity type that implements the `RootInfo` delegate. Each `RootInfo` entity instance encapsulates a summary of one archived entity.

In ClaimCenter, the archive root info entity is always a `ClaimInfo` entity. The `ClaimInfo` entity describes a claim, including its archiving status. The interface type `RootInfo`, in APIs or in documentation, always refers to `ClaimInfo`. Your plugin implementation must store enough reference information in the `ClaimInfo` entity type to determine how to retrieve any archived document from the archive backing store.

Changes to ID values

It is possible for the `ID` value of the restored `Claim` entity to be different from that of the original archived entity. However, in all cases, the `PublicId` value remains the same. Never attempt to use the entity `ID` value to identify an archive entity during the archive storage or retrieval process. Instead, use the `PublicId` value to identify an entity.

Encrypting archival data

In the base configuration, Guidewire provides the means to encrypt ClaimCenter data through the use of encryption plugins that implement the `gw.plugin.util.IEncryption` interface. An encryption plugin encrypts data fields (table columns) that you tag for encryption in the data model.

Each encryption plugin implementation must return a unique ID string from its `getEncryptionIdentifier` method. The archival XML document contains the ID of the encryption plugin used to encrypt its tagged fields. Configuration parameter `DefaultXmlExportEncryptionId` sets the ID of the encryption plugin to use to encrypt the XML data fields, both during the archive process and during the export of XML data. You can use the ID of any existing encryption plugin for this purpose or create a custom encryption implementation with its own ID specifically for exporting XML. The plugin implementation that you use for encryption must remain registered as long as there are

archive documents with that key, meaning that the XML documents that use that key have not been purged from the archive store.

If you want to encrypt more than just the fields tagged as encrypted, leave the value of `DefaultXmlExportIEncryptionId` empty and implement any strategy that you like for protecting sensitive information. You will have to devise your own strategy also if there are no fields tagged as encrypted or if not all of the sensitive fields are tagged.

In working with archival data, Guidewire recommends the following:

- Use a secure communication channel such as TLS in storing and retrieving the archival XML.
- Encrypt the entire XML document at rest using the facilities of the chosen storage system.

For an example of how to create an encryption class, see Gosu class `PBEEncryptionPlugin` in the following Guidewire package:

```
gw.plugin.encryption.impl
```

See also

- For general information on encryption, see “Encryption integration” on page 263.
- For information on how to enable the `IEncryption` plugin, see “Enable your encryption plugin implementation” on page 271.
- For information on how to tag a data field for encryption, see “Setting encrypted properties” on page 263.

Archive source plugin methods and archive transactions

It is important to understand that ClaimCenter calls some Archive Source plugin methods inside an archive transaction and other methods outside a transaction. The behavior varies by plugin method. Be careful to understand any changes to database data outside the transaction, especially with respect to any error conditions.

For example, ClaimCenter always calls the plugin method “`storeFinally`” on page 660 outside the main database transaction for the storage request. If the storage request fails, ClaimCenter does not commit the data changes to the database. However, any changes you make during the call to `storeFinally` do persist to the database.

Important considerations for archiving

1. Be extremely careful that you understand potential failure conditions and the relationship between each archive method and associated database transactions.
2. Do not generate event messages in the Event Fired rules for entity updates in the archive Transaction bundle. These messages cause the archive transaction to fail.

See also

- *Configuration Guide*

Archive source plugin storage methods

ClaimCenter calls various `IArchiveSource` methods during the archiving process. The following list of methods defines the main storage methods:

- `prepareForArchive`
- `store`
- `storeFinally`
- `updateInfoOnStore`

Your plugin implementation must implement all of these methods. Refer to the demonstration plugin for guidance.

Note: In the following method declarations, the interface type `RootInfo` always refers to a `ClaimInfo` entity instance.

prepareForArchive

The `IArchiveSource.prepareForArchive` method has the following signature.

```
prepareForArchive(info : RootInfo)
```

The method call for `prepareForArchive` occurs outside the archive transaction. As such:

- ClaimCenter does not roll back any changes if the archiving operation fails.
- ClaimCenter does not commit changes automatically if the archiving operation succeeds.

You use this method for the (somewhat unusual) case in which you want to prepare some data regardless of whether a domain graph instance actually archives successfully. The method has no transaction of its own. If you want to update data, then you must create a bundle and commit that bundle yourself.

In the demonstration plugin implementation, the method body is empty. You must provide your own implementation of this plugin method.

store

The `IArchiveSource.store` method has the following signature.

```
store(graph : InputStream, info : RootInfo)
```

The `store` method is the main configuration point for taking XML data (in the form of an `InputStream` object) and transferring it to an archive backing store.

ClaimCenter calls the `store` method inside the archiving transaction, after deleting rows from the database, but before performing the database commit. Your implementation of this method must store the archive XML document.

During the method call, the archiving process passes in the `java.io.InputStream` object that contains the generated XML document. This is the data that your Archive Source plugin must send to the archive backing store.

The archiving process passes in an instance of the `RootInfo` entity to the plugin so that it can insert or update additional reference information to help with restore. Generally speaking, this is not the best place to make changes to the `RootInfo` entity instance. Usually, it is best to make those changes in the `updateInfoOnStore` method.

Generally speaking, it is best to change no entity data at all in the `store` method. Within the `store` method (or any other part of the data base transaction), the only properties that are safe to change are the non-array properties on the `RootInfo` entity. The only safe reason to change entity data in the `store` method is to add a unique retrieval ID to an extension property on the `RootInfo` entity. Only make this change from the `store` method if your archive backing store requires this data for retrieval and that ID is only available after sending data.

If your plugin is unable to store the XML document, then ClaimCenter expects the plugin to throw an error. ClaimCenter treats this error as a storage failure and rolls back the transaction. The transaction rollback also rolls back any changes to entity instances that you set up in your `updateInfoOnStore` method call.

storeFinally

The `IArchiveSource.storeFinally` method has the following signature.

```
storeFinally(info : RootInfo, finalStatus : ArchiveFinalStatus, cause : List<String>)
```

ClaimCenter calls this method outside the archiving transaction after completing that transaction. The value of the `finalStatus` parameter indicates whether the archiving delete operation was successful. In the rare case in which the delete transaction fails, it is possible to reverse any changes that were not part of the transaction.

To change any data, you must perform any entity instance modification within a call to `Transaction.RunWithNewBundle(block)`. See the *Gosu Reference Guide*.

If ClaimCenter calls this method with errors, the value of `finalStatus` is something other than `success`. The `cause` parameter contains a list of `String` objects that describe the cause of any failures. The `String` objects are the text for the exception `cause` hierarchy.

This permits two different design strategies:

- The recommended technique is to send and commit your data in the external source in the `store` method. If there are failure conditions, you can use your implementation of the `storeFinally` method to back out of any changes in the external data store. For example, you can delete any temporary files that the archive process created. The demonstration implementation for this plugin illustrates this process.
- Alternatively, you can perform a two-stage commit. In other words, send the data to the external system in the `store` method, but finalize the data in the `storeFinally` method if and only if parameter `finalStatus` indicates success.

It is important to be careful about what kinds of work you perform in the `storeFinally` method to properly handle error conditions. If the `storeFinally` method throws an exception, the application logs the exception, but ClaimCenter has already completed the main database transaction. Be sure to carefully catch any exception and handle all error conditions. There is no rollback or recovery that the application can perform if `storeFinally` does not complete its actions due to errors or exceptions within the `storeFinally` implementation.

[updateInfoOnStore](#)

The `IArchiveSource.updateInfoOnStore` method has the following signature.

```
updateInfoOnStore(info : RootInfo)
```

Whenever ClaimCenter receives a request to archive a `RootInfo` object, it first checks whether the entity instance is eligible for archiving. After these checks runs, but before ClaimCenter deletes the entity information from the database, it calls an internal method, `updateInfoOnArchive`, to prepare the entity for archiving. This internal method in turn calls the `updateInfoOnStore` method.

ClaimCenter calls the `updateInfoOnStore` method after the application prepares the data for archive but before calling the `store` method. Despite the name of the `updateInfoOnStore` method, this method is not the only place to modify the `RootInfo` entity. See the discussion on the plugin method `store`.

ClaimCenter calls the `updateInfoOnStore` method inside the archiving transaction. This enables you to make additional updates on `RootInfo` entities. For example, it is possible to use this method to write logic to update calculated fields on the `ClaimInfo` entity that ClaimCenter uses for aggregate reports or searches.

In ClaimCenter, the default plugin implementation of this method calls the following additional plugin methods:

[removeMessageHistoriesForClaim](#)

Finds all the message histories associated with the specified claim instance and removes them.

[clearPolicyLocationSummaryJoinEntry](#)

Removes all references to `PolicyLocationSummaryJoin` from the archive.

WARNING: Do not create custom code that generates update events for any entity type that can possibly be part of the archive bundle. Any active event messages in the archive transaction bundle cause archiving to fail.

Depending on the kind of action undertaken in `updateInfoOnStore`, you might need to perform similar or complementary changes in plugin methods `updateInfoOnRetrieve` and `updateInfoonDelete`.

WARNING: Only modify the `RootInfo` entity (`ClaimInfo`) in the `updateInfoOnStore` method. It is dangerous and unsupported to modify any other entity instances in the claim graph in this method.

See also

- [Configuration Guide](#)

Archive source plugin retrieval methods

ClaimCenter calls the following `IArchiveSource` methods during the archive retrieval lifecycle:

- `retrieve`
- `retrieveFinally`

- `updateInfoOnRetrieve`

Your plugin implementation must implement all of these methods. Refer to the demonstration plugin for guidance.

Note: In the following method declarations, the interface type `RootInfo` always refers to a `ClaimInfo` entity instance.

`retrieve`

The `IArchiveSource.retrieve` method has the following signature.

```
retrieve(info : RootInfo) : InputStream
```

This method is the main configuration point for retrieving XML data from the archive backing store. You must return the data as an `InputStream` object. The archiving process passes the `RootInfo` entity instance to the plugin method to assist your plugin implementation to identify the correct data in the external system.

For archive retrieval to work correctly, the `RootInfo` object must store enough information to determine how to retrieve the XML document from the archive backing store.

`retrieveFinally`

The `IArchiveSource.retrieveFinally` method has the following signature.

```
retrieveFinally(info : RootInfo, finalStatus : ArchiveFinalStatus, cause : List<String>)
```

The retrieval process calls this method as the last retrieval step, outside of the retrieve transaction. The value of the `finalStatus` parameter indicates whether the retrieve was successful. Use this plugin method to perform other actions on the storage.

It is up to you to decide what additional handling the retrieved data requires, if any. For example, you can use this method to perform final clean up on files in the archive backing store.

IMPORTANT: Guidewire does not recommend, nor does Guidewire support, the use of this method to make changes to data after you retrieve the data. Any attempt to commit these changes in the `retrieveFinally` method invokes the Preupdate and Validation rules. This is undesirable and unsupported. As ClaimCenter commits the main transaction for the retrieve process, it does not run these rules. Therefore, it is possible, to retrieve entity instances that do not pass validation rules, for example. Committing additional changes in `retrieveFinally` could fail validation, causing only those changes to be rolled back, not the entire retrieve request.

`updateInfoOnRetrieve`

The `IArchiveSource.updateInfoOnRetrieve` method has the following signature.

```
updateInfoOnRetrieve(info : RootInfo)
```

The retrieval process calls this method within the retrieval transaction. This occurs after the archiving process populates the bundle of the `RootInfo` entity with the entities produced by parsing the XML returned by the earlier call to `retrieve(RootInfo)`.

From within the `updateInfoOnRetrieve` method, you can modify the new entity instances in the archive domain graph as well as the `RootInfo` entity instance. For example, you can reset the date on which an entity type is eligible for archiving again.

Archive source plugin utility methods

ClaimCenter calls the following `IArchiveSource` utility methods during the archive lifecycle as needed:

- `delete`
- `getStatus`
- `handleUpgradeIssues`

- refresh
- retrieveSchema
- setParameters
- storeSchema
- updateInfoonDelete

Your plugin implementation must implement all of these methods. Refer to the demonstration plugin for guidance.

Note: In the following method declarations, the interface type `RootInfo` always refers to a `ClaimInfo` entity instance.

delete

The `IArchiveSource.delete` method has the following signature.

```
delete(info : RootInfo)
```

Your implementation of this method must delete the data identified by the specified `RootInfo` entity from the archive backing store. ClaimCenter calls the `delete` method during the purge process.

IMPORTANT: In your implementation of the `delete` method, you must delete any documents and associated document metadata linked to the archived entity instance.

The delete process calls this method after ClaimCenter deletes associated entities returned from `updateInfoonDelete`.

getStatus

The `IArchiveSource.getStatus` method has the following signature.

```
getStatus()
```

The `getStatus` method returns an object that implements the `ArchiveSourceInfo` interface. The `ArchiveSourceInfo` object has the following methods.

Method	Returns
<code>getAsOf</code>	Returns the date on which ClaimCenter generated the status information.
<code>getDeleteStatus</code>	Returns a typekey from the <code>ArchiveSourceStatus</code> typelist. The typecode is one of the following:
<code>getRetrieveStatus</code>	<ul style="list-style-type: none">• Available - Archiving service is available.• Failure - Last attempt to archive failed.
<code>getStoreStatus</code>	<ul style="list-style-type: none">• Manually flagged - Archiving service was manually flagged as unavailable.• Not Configured - Archiving service not configured.• Not Enabled - Archiving service not enabled.• Not Started - Archiving service not started.• Queue Available - Archiving service is not available. But, it is possible to add the request to the archiving queue.

To retrieve a status value, use syntax similar to the following, with XXX being either `Delete`, `Retrieve`, or `Store`.

```
ArchiveSourceInfo.getStatus().getAsOf()  
ArchiveSourceInfo.getStatus().getXXXStatus()
```

handleUpgradeIssues

The `IArchiveSource.handleUpgradeIssues` method has the following signature.

```
handleUpgradeIssues(info : RootInfo, root : KeyableBean, issues : List<Issue>)
```

Use this method to mange a set of passed-in issues that occurred during an archive operation. In the base configuration, the default implementation provides the means to associates a set of archive failure issues with the archive Claim entity.

[refresh](#)

The `IArchiveSource.refresh` method has the following signature.

```
refresh()
```

In the base configuration, this method sets a number of archive-related parameter values. Archive source plugin method `setParameters` calls this `refresh` method.

[retrieveSchema](#)

The `IArchiveSource.retrieveSchema` method has the following signature.

```
retrieveSchema(platformMajor : int, platformMinor : int, appMajor : int, appMinor : int,  
extension : int) : InputStream
```

This method retrieves the XSD associated with a `data model + version` number combination as a `FileInputStream` object. The XSD describes the format of the XML files for that version.

In the base configuration, the default implementation class for the archive source plugin (`ClaimInfoArchiveSource`) prints the absolute path for the XSD schema files to the application console during server start.

In the base configuration, the default implementation class for the archive source plugin (`ClaimInfoArchiveSource`) prints the absolute path for the XSD schema files to the application console during server start.

[setParameters](#)

The `IArchiveSource.setParameters` method has the following signature.

```
setParameters(parameters : Map)
```

In the base configuration, this method prints a warning in the archive log if the server is in production mode. The warning indicates that the default sample implementation of the `ClaimInfoArchiveSource` plugin is not suitable for production environments.

In addition, this method calls the plugin class “refresh” on page 664 method, which resets a number of archive-related parameter values.

[storeSchema](#)

The `IArchiveSource.storeSchema` method has the following signature.

```
storeSchema(platformMajor : int, platformMinor : int, appMajor : int, appMinor : int,  
extension : int, schema : InputStream)
```

This method stores the XSD associated with the specified `data model + version number` combination. The XSD describes the format of the XML files for that version.

In the base configuration, the default implementation class for the archive source plugin (`ClaimInfoArchiveSource`) prints the absolute path for the XSD schema files to the application console during server start.

[updateInfoonDelete](#)

The `IArchiveSource.updateInfoonDelete` method has the following signature.

```
updateInfoonDelete(info : RootInfo) : List<Pair<IEntityType, List<Key>>>
```

At some later time after archiving, it is possible for a user to decide to delete the archived entity instance entirely, including the data that remained in the application database. Guidewire calls this process *purgung*. During data purging, it is possible to delete the archival data from the archive backing store, but Guidewire does not require that you do so.

During purging, ClaimCenter calls the `updateInfoonDelete` method to determine additional entity instances that ClaimCenter must delete if it deletes the entity instance represented by `RootInfo`. ClaimCenter never calls this method

during the main archiving step. The delete process calls this method within the transaction in which it deletes the `RootInfo` entity instance and related entities from the active database.

If your implementation of the `updateInfoOnStore` method creates extension entity instances that link to the `RootInfo` entity type, return these entity instances using plugin method `updateInfoonDelete` during a delete operation.

WARNING: Declaring these relationships in the `updateInfoonDelete` method is important because you might have created other entities that ClaimCenter cannot discover by looking at the foreign key references on the `RootInfo` entity.

The return type is a list of pair (`Pair`) objects. Each pair object is parameterized on the following:

- An entity type
- A list of `Key` objects. The `Key` object contains the relevant `object.ID` value for that object.

Typically, it is inappropriate to make changes to entity instances in this method. If this method does make changes, the application commits those changes before calling plugin method “delete” on page 663.

If you create entities in the `updateInfoOnStore` method that link to the `RootInfo` entity, return those entities in the return results of this method. Generally speaking, if you mark an entity to delete by returning it from this method, you must return also all entities that link to those entities.

The order of the types and IDs in the list is important. Deletion happens in the order in the list and deletion is permanent. If entity type A links to entity type B, which links to the `RootInfo` entity type, you must return A before B.

Typically you would determine the correct order of types and then delete all entities of that entity type all at once.

However, although usually unnecessary, Guidewire does support the option to return an entity type more than once in the return list. Theoretically in complex configuration edge cases, it is possible that this is necessary to enforce proper ordering of the deletion process. For example, this allows you to enforce the following deletion order:

1. Delete entity instance with ID 100 of type Type1.
2. Delete entity instance with ID 900 of type Type2.
3. Delete entity instance with ID 101 of type Type1, a type already mentioned.

Servlets

A *servlet* is a small program that runs on a web server to process and answer client requests. You can define Gosu classes as simple web servlets inside your ClaimCenter application. By using a servlet, you can define HTTP entry points to custom code. Use this technique to define arbitrary Gosu code that a user or tool can call from a configurable URL.

You use the `@Servlet` annotation in your Gosu class to identify the class as a servlet. This annotation specifies the servlet query path in the URL that launches the servlet. To specify the query servlet path, you can use a static `String` value or a pattern described by a Gosu block. The URL for a servlet is the base URL for your ClaimCenter application, followed by `/service`, and finally the servlet query path. The following line shows this syntax:

```
http://serverName:8080/cc/service/servletQueryPath
```

You register your servlet with ClaimCenter by adding an entry in the file `ClaimCenter/configuration/config/servlet/servlets.xml`. ClaimCenter determines from the URL the servlet that owns the HTTP request.

Servlets are separate from web services that use the SOAP protocol. Servlets provide no built-in object serialization or deserialization, such as the SOAP protocol supports. Servlets are also separate from the Guidewire PCF entry points feature.

The ClaimCenter application provides an informational servlet that lists the available servlets. In the base configuration, this servlet is disabled if the ClaimCenter application is in production mode. Use the following URL to display the list:

```
http://serverName:8080/cc/service/info
```

Viewing servlet analysis information

Guidewire provides a means to view servlet activity in ClaimCenter Profiler. To view this information, navigate to the following location in ClaimCenter Server Tools:

Guidewire Profiler > Configuration

For information on how to use Guidewire Profiler, see the *Administration Guide*.

Implementing servlets

The ClaimCenter servlet implementation uses standard Java classes in the package `javax.servlet.http` to define the servlet request and response. The base class for your servlet must extend `javax.servlet.http.HttpServlet` directly, or extend a subclass of `HttpServlet`. The typical servlet methods that you implement have

`javax.servlet.http.HttpServletRequest` and `javax.servlet.http.HttpServletResponse` objects as parameters.

Extending the base class `javax.servlet.http.HttpServlet` provides no inherent security for the servlet. By default, users can trigger servlet code without authenticating. Accessing ClaimCenter without authentication is a security risk in a production system.

WARNING: You must add your own authentication system for servlets to protect information and data integrity. If you have any questions about server security, contact Guidewire Customer Support.

ClaimCenter includes abstract classes that you can extend to provide authentication in the `gw.servlet` package. If you need your servlet to perform tasks other than simple authentication, you can use static methods on the utility class, `gw.servlet.ServletUtils`.

During the development of your servlet, you can send debug messages to the Studio console by using the `print` function. When your development is complete, you can use a logger to track information about the servlet usage. You retrieve the appropriate logger by calling the `forCategory` method in the `gw.api.system.CCLoggerCategory` class.

See also

- “Implementing servlet authentication” on page 672

@Servlet annotation

You must pass arguments to the `@Servlet` annotation to specify the URLs that your servlet handles. You provide a search pattern to test against the *servlet query path*. The servlet query path is every character after the computer name, the port, the web application name, and the word “/service”. The servlet query path must begin with a slash (/) character. Make sure that you do not create ambiguous servlet query paths by using patterns that match the same strings.

Multiple signatures of the annotation constructor support different use cases.

- `@Servlet(pathPattern : String)`

Use this annotation constructor syntax for straightforward pattern matching, with less flexibility than full regular expressions.

This annotation constructor declares the servlet URL pattern matching to use Apache `org.apache.commons.io.FilenameUtils` wildcard syntax. Apache `FilenameUtils` syntax supports Unix and Windows file names with limited wild card support for ? (single character match) and * (multiple character match) similar to DOS file name syntax. The syntax also supports the Unix meaning of the ~ symbol. Matches are always case sensitive.

- `@Servlet(pathMatcher : block(path : String) : boolean)`

Use this annotation constructor syntax for highly flexible pattern matching.

You provide a Gosu block that can do arbitrary matching on the servlet query path. The Gosu block takes a `String` parameter, which is the servlet query path. Write the block to return `true` if and only if the `String` matches the URLs that your servlet handles. You can use methods on the `String` parameter to perform simple or more complex pattern matching.

For example, you can use the `startsWith` method of the `String` argument. The following example servlet responds to URLs that start with the text “/test” in the servlet query path:

```
@Servlet(\ path : String -> path.startsWith("/test"))
```

This example servlet responds to any of the following URLs:

```
http://localhost:8080/cc/service/test  
http://localhost:8080/cc/service/tester  
http://localhost:8080/cc/service/test/more
```

You can use other methods to do more flexible pattern matching, such as full regular expressions. Test your regular expressions thoroughly. For some patterns, not all matching values are valid servlet query paths. The following example interprets regular expressions by using the `matches` method of the `String` argument:

```
@Servlet(\ path : String -> path.matches("/test([0-9](/.*))?"))
```

This example servlet responds to URLs that start with the text "/test" in the servlet query path, and optionally a digit and a slash followed by other text:

- A single page URL:

```
http://localhost:8080/cc/service/test  
http://localhost:8080/cc/service/test0
```

- An entire virtual file hierarchy, such as:

```
http://localhost:8080/cc/service/test1/another/level
```

To match multiple page URLs that are not described in traditional hierarchies as a single root directory with subdirectories, you could intercept URLs with the regular expression:

```
@Servlet(\ path : String -> path.matches("(/.*)?/subfolder_one_level_down")
```

That pattern matches all of the following URLs:

```
http://localhost:8080/cc/service/test1/subfolder_one_level_down  
http://localhost:8080/cc/service/test2/subfolder_one_level_down  
http://localhost:8080/cc/service/test3/subfolder_one_level_down
```

See also

- For full documentation on this `String` regular expressions, refer to this Oracle Javadoc page:
<https://docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html#sum>

Servlet definition in `servlets.xml`

The `configuration/config/servlet/servlets.xml` file contains definitions for Gosu servlets. You add definitions for your custom servlets to this file. You define a custom servlet in a `<servlet>` element.

The following attributes are available to the `<servlet>` element.

`class`

Required. The fully qualified name of the class that implements the servlet.

`env`

Optional. A comma-separated list of names of environments for which the servlet definition applies.

`info-description`

Optional. The description of the servlet.

`info-path`

Optional. Reserved for future use.

`server`

Optional. The server for which the servlet definition applies. The value of this attribute is either a server role preceded by the # character or a server ID.

The `<servlet>` element supports an optional `<param>` subelement for each configuration parameter. The names and values of the parameters appear when the `InfoServlet` servlet runs if the value of its `ExposeDetails` parameter is `true`. The following attributes are available to the `<param>` subelement.

`name`

Required. The name of the parameter.

value

Required. The value for the parameter that ClaimCenter uses at start up.

server

Optional. Specifies the server for which the parameter definition applies. The value of this attribute is either a server role preceded by the # character or a server ID.

env

Optional. A comma-separated list of names of environments for which the parameter definition applies.

Example

The configuration/config/servlet/servlets.xml file contains the following servlet definition, which demonstrates many of these elements and attributes.

```
<servlet
  class= "com.guidewire.pl.system.servlet.InfoServlet"
  info-description="This servlet will dump the information about its state and its servlets it presents. If
ExposeDetails it will also dump the parameters"
  info-path="info">
<param
  name="ExposeDetails"
  value="false"/>
<param
  env="detail"
  name="ExposeDetails"
  value="true"/>
<param
  name="ExposeInProduction"
  value="false"/>
</servlet>
```

Important HttpServletRequest object properties

The following list shows some important properties on the request object. The example values for these properties relate to the following URL:

```
http://localhost:8080/cc/service/test0?abc
```

RequestURL

The URL that the client used to make the request

Example value: `http://localhost:8080/cc/service/test0`

RequestURI

The part of this request's URL from the protocol name up to the query string in the first line of the HTTP request

Example value: `/cc/service/test0`

QueryString

The query string in the request URL after the servlet query path and the ? character, if present

Example value: `abc`

PathInfo

Any extra path information associated with the URL that the client sent when it made this request

Example value: `/test0`

HeaderNames

All the names of the request headers as an Enumeration of String objects

Not all servlet containers provide the value of this property.

See also

- For full documentation on this class, refer to this Oracle Javadoc page:

<http://docs.oracle.com/javaee/7/api/javax/servlet/http/HttpServletRequest.html>

- For full documentation on the `HTTPServletResponse` class, refer to this Oracle Javadoc page:
<http://docs.oracle.com/javaee/7/api/javax/servlet/http/HTTPServletResponse.html>

Create and test a basic servlet

A basic servlet displays a message in ClaimCenter. You call this servlet in the context of ClaimCenter.

About this task

For simplicity, this basic servlet overrides the `service` method of the `HttpServlet` class. A production servlet can override other methods of this class, such as `doGet` or `doPost`.

Procedure

1. Write a Gosu class that extends the class `javax.servlet.http.HttpServlet`.

For this example, create a class in **configuration > gsrc > mycompany > test > servlets**. Make the name of the class `TestingServlet`.

2. On the line before your class definition, add the `@Servlet` annotation to specify the servlet query path for this servlet:

```
@Servlet( \ path : String ->path.matches("/test(/.*)?") )
```

If the `Servlet` text appears red in the Studio editor, press **Alt+Enter** to import the `gw.servlet.Servlet` class.

3. Override the `service` method to do your actual work. Your `service` method takes a servlet request object (`HttpServletRequest`) and a servlet response object (`HttpServletResponse`) as parameters.

Press **Alt+Insert** in the Studio editor to select the correct method to override.

4. In your `service` method, specify the work that the servlet request object performs.

WARNING: You must add an authentication system to your servlets to protect information and data integrity. If you have questions about server security, contact Guidewire Customer Support.

In your `service` method, remove any template code and write an HTTP response using the servlet response object. The following simple response sets the content MIME type and the status of the response (OK):

```
resp.ContentType = "text/plain"  
resp.setStatus(HttpServletRequest.SC_OK)
```

To write output text or binary data, use the `Writer` property of the response object.

```
resp.getWriter.append("I am the page " + req.getPathInfo)
```

5. Register the servlet class in the following file:

```
ClaimCenter/configuration/config/servlet/servlets.xml
```

Add one `<servlet>` element that references the fully qualified name of your class.

```
<servlets  
    xmlns="http://guidewire.com/servlet">  
    <servlet  
        class="mycompany.test.servlets.TestingServlet"/>  
</servlets>
```

6. Start the QuickStart server in Guidewire Studio.

If the server is already running, stop and restart the server.

7. Test the servlet at the URL:

```
http://localhost:8080/cc/service/test
```

The text "/test" in the URL is the part that matches the servlet query path. The `TestingServlet` class specifies that path in the `@Servlet` annotation. If necessary, change the port number and the server name to match your ClaimCenter application.

The following text appears on the page:

```
I am the page /test
```

See also

- “Example of a basic servlet” on page 672
- “Implementing servlet authentication” on page 672

Example of a basic servlet

The following simple Gosu servlet runs in the context of the ClaimCenter application, but provides no authentication. For a production servlet, you must add authentication. This example servlet responds to URL substrings that start with the string `/test`. If an incoming URL matches that pattern, the servlet displays the `PathInfo` property of the response object, which contains the path.

```
package mycompany.test.servlets

uses gw.servlet.HttpServlet
uses javax.servlet.http.HttpServletRequest
uses javax.servlet.http.HttpServletResponse
uses javax.servlet.http.HttpServlet

@WebServlet( \ path : String ->path.matches("/test(/.*)?"))
class TestingServlet extends HttpServlet {

    override function service(req: HttpServletRequest, response: HttpServletResponse) {
        // ** SECURITY WARNING - FOR REAL PRODUCTION CODE, ADD AUTHENTICATION CHECKS HERE!

        // Trivial servlet response to test that the servlet responds
        response.ContentType = "text/plain"
        response.setStatus(HttpServletRequest.SC_OK)
        response.getWriter.append("I am the page " + req.PathInfo)
    }
}
```

For ClaimCenter to provide access to the servlet, the `ClaimCenter/configuration/config/servlet/servlets.xml` file contains the following information.

```
<servlets
    xmlns="http://guidewire.com/servlet">
    <servlet
        class="mycompany.test.servlets.TestingServlet"/>
</servlets>
```

See also

- “Implementing servlet authentication” on page 672

Implementing servlet authentication

Extending the base class `javax.servlet.http.HttpServlet` provides no inherent security for your servlet. By default, anyone can trigger servlet code without authenticating.

WARNING: You must add your own authentication for your servlet to protect information and data integrity. If you have any questions about server security, contact Guidewire Customer Support.

The package `gw.servlet` provides abstract classes that you extend to create a servlet that provides user authentication. The class `AbstractGWAAuthServlet` translates the security headers in the request and authenticates with the Guidewire server. The subclass `AbstractBasicAuthenticationServlet` authenticates using HTTP basic authentication. These classes support using only one type of authentication at run time. To support both HTTP basic authentication and Guidewire authentication in the same servlet, extend `HTTPSServlet` and use utility methods from the `ServletUtils`

class. By using `ServletUtils`, you can use the session key if available and if not you can use HTTP Basic authentication headers or custom headers.

For security reasons, a servlet saves the connection's session ID only if the HTTP connection is secure (HTTPS). This behavior can be changed for network topologies that secure the connection through other means, such as by not allowing external access to the server. To force the saving of the connection's session ID, set the system property `gw.servlet.ServletUtils.BypassIsSecure` to true.

Guidewire recommends that your servlets use HTTP basic authentication, which is supported by the `AbstractBasicAuthenticationServlet` class.

Create a servlet that provides basic authentication

You can write a servlet that uses basic authentication to log in to ClaimCenter. You test this servlet in another procedure.

Procedure

1. Write a Gosu class that extends the class `gw.servlet.AbstractBasicAuthenticationServlet`.
2. Add the `@Servlet` annotation on the line before your class definition:

```
@Servlet( \ path : String ->path.matches("/test(.*)?"))
```

3. Override the applicable REST method, for example, the `doGet` method, to do your actual work. Your method takes a servlet request object (`HttpServletRequest`) and a servlet response object (`HttpServletResponse`) as parameters.
4. In your method, determine what work to do using the servlet request object.

In your `doGet` method, write an HTTP response using the servlet response object. The following simple response sets the content MIME type and the status of the response (OK):

```
resp.ContentType = "text/plain"
resp.setStatus(HttpServletRequest.SC_OK)
```

To write output text or binary data, use the `Writer` property of the response object.

```
resp.getWriter.append("I am the page " + req.getPathInfo)
```

5. Register the servlet class in the file:

```
ClaimCenter/configuration/config/servlet/servlets.xml
```

Add one `<servlet>` element that references the fully qualified name of your class:

```
servlets xmlns="http://guidewire.com/servlet">
  <servlet class="mycompany.test.servlets.TestingServlet"/>
servlets
```

See also

- “Example of a basic authentication servlet” on page 674

Test your basic authentication servlet

A servlet can use basic authentication to log in to ClaimCenter. This procedure shows you how to test a basic authentication servlet.

Before you begin

This topic uses the `mycompany.test.TestingServlet` servlet, as shown in “Example of a basic authentication servlet” on page 674. To test the servlet, you must use a tool that supports adding HTTP basic authentication headers for the user name and password.

Procedure

- Run the QuickStart server with the following command:

```
gwb runServer
```

- Test the servlet at the URL:

```
http://localhost:8080/cc/service/test
```

The text "/test" in the URL is the part that matches the servlet string. Change the port number and the server name to match your ClaimCenter application.

- In the authentication dialog box that appears, type valid login credentials.

The following text appears on the page:

```
I am the page /test
```

The following messages appear in the console in Studio:

```
servlet test url: /cc/service/test
query string: null
```

- Test additional pages:

```
http://localhost:8080/cc/service/test/is/this/working?param=value
```

The following text appears on the page:

```
I am the page /test/is/this/working
```

The following messages appear in the console in Studio:

```
servlet test url: /cc/service/test
query string: param=value
```

- To test the HTTP basic authentication without using the default login dialog:

- a) Close your browser to remove the session context.
- b) Re-test your servlet, using a tool that supports adding HTTP headers for HTTP basic authentication.

Example of a basic authentication servlet

The following Gosu servlet runs in the context of the ClaimCenter application, and uses HTTP basic authentication. This example servlet responds to URL substrings that start with the string /test. If an incoming URL matches that pattern, the servlet displays the PathInfo property of the response object, which contains the path. This servlet also sends debug messages to the Studio console when you run the Quickstart server.

```
package mycompany.test.servlets

uses gw.servlet.HttpServlet
uses gw.servlet.AbstractBasicAuthenticationServlet

uses javax.servlet.http.HttpServletRequest
uses javax.servlet.http.HttpServletResponse

@Servlet( \ path : String ->path.matches("/test(/.*)?"))
class TestingServlet extends gw.servlet.AbstractBasicAuthenticationServlet {

    override function doGet(req: HttpServletRequest, resp : HttpServletResponse) {
        print("servlet test url: " + req.RequestURI)
        print("query string: " + req.QueryString)

        resp.ContentType = "text/plain"
        resp.setStatus(HttpServletResponse.SC_OK)
        resp.getWriter.append("I am the page " + req.PathInfo)
    }
}
```

```
override function isAuthenticationRequired( req: HttpServletRequest ) : boolean {  
    // -- TODO -----  
    // Read the headers and return false if user is already authenticated  
    // -----  
    return true  
}
```

This servlet responds to URLs with the word test in the service query path, such as the URL:

```
http://localhost:8080/cc/service/test/is/this/working
```

Supporting multiple authentication types

The `gw.servlet.ServletUtils` class provides methods that support testing for existing authenticated sessions. A typical design pattern is to first call the `getAuthenticatedUser` method to test whether there is an existing session token that represents valid credentials. If the `getAuthenticatedUser` method returns `null`, you can attempt to use HTTP basic authentication by calling the method `getBasicAuthenticatedUser`.

In a single sign-on environment, you get the user from the current HTTP session before calling `login`. If the session does not have a valid service token, you do HTTP basic authentication. Performing calls in this order ensures that you use existing single sign-on credentials and do not terminate an existing active session or cause a user to log in multiple times.

Abstract HTTP basic authentication servlet class

The `gw.servlet.AbstractBasicAuthenticationServlet` class extends `AbstractGWAAuthServlet` to support HTTP Basic authentication.

Your main task is to override the method for the HTTP request that your servlet handles to do your required work. ClaimCenter authenticates using the HTTP basic authentication headers before calling this method.

Also, override the `isAuthenticationRequired` method and return `true` if this request requires authentication. You can use methods on the `ServletUtils` class to check the current authentication status of the session.

You can implement the following HTTP methods in your servlet class that extends the `AbstractBasicAuthenticationServlet` class:

- `doDelete` – Handles an HTTP DELETE request.
- `doGet` – Handles an HTTP GET request.
- `doPost` – Handles an HTTP POST request.
- `doPut` – Handles an HTTP PUT request.

For full documentation on the `HttpServletRequest` class, refer to this Oracle Javadoc page:

<http://docs.oracle.com/javaee/7/api/javax/servlet/http/HttpServletRequest.html>

Abstract Guidewire authentication servlet class

IMPORTANT: Guidewire recommends that your servlets use HTTP basic authentication, which is supported by the `gw.servlet.AbstractBasicAuthenticationServlet` class.

To use the session key created from a Guidewire application that shares the same application context, you can write your servlet to extend the class `gw.servlet.AbstractGWAAuthServlet`. You can use methods on the `ServletUtils` class to check the current authentication status of the session. You must override the following methods:

- `service` – Your main task is to override the `service` method to do your required work. To check the HTTP request type, call the `getMethod` method on the servlet request object. ClaimCenter already authenticates the session key if it exists before calling your method.

- `authenticate` – Create and return a session ID.
- `storeToken` – You can store the session token in this method, or you can leave your method implementation empty.
- `invalidAuthentication` – Return a response for invalid authentication. For example:

```
override function invalidAuthentication( req: HttpServletRequest,
                                         resp: HttpServletResponse ) : void {
    resp.setHeader( "WWW-Authenticate", "Basic realm=\"Secure Area\"")
    resp.setStatus( HttpServletResponse.SC_UNAUTHORIZED )
}
```

ServletUtils authentication methods

ClaimCenter includes a utility class `gw.servlet.ServletUtils` that you can use in your servlet to enforce authentication. The three methods in the `ServletUtils` class each correspond to a different source of authentication credentials. The following table summarizes each `ServletUtils` method. In all cases, the first argument is a standard Java `HttpServletRequest` object, which is an argument to your main servlet method `service` or a REST method such as `doGet`.

Source of credentials	ServletUtils method name	Description	Method arguments
Existing ClaimCenter session	<code>getAuthenticatedUser</code>	<p>If this servlet shares an application context with a running Guidewire application, there may be an active session token. If a user is currently logged in to ClaimCenter, this method returns the associated <code>User</code> object.</p> <p>Always check the return value. The method returns <code>null</code> if authentication failed. Reasons for authentication failure include:</p> <ul style="list-style-type: none"> • There is no active authenticated session with correct credentials. • The user exited the application. • The session ID is not stored on the client. • The session <code>ServiceToken</code> timeout has expired. 	<ul style="list-style-type: none"> • <code>HttpServletRequest</code> object • A Boolean value that specifies whether to update the date and time of the session
HTTP Basic authentication headers	<code>getBasicAuthenticatedUser</code>	<p>If there is no active session, you can use HTTP basic authentication. This method gets the appropriate HTTP headers for name and password and attempts to authenticate. You can use this type of authentication even if there is an active session. This method forces creation of a new session. The method gets the headers to find the user name and password and returns the associated <code>User</code> object.</p> <p>Always check the return value. The method returns <code>null</code> if authentication failed.</p> <p>For login problems, this method might throw the exception <code>gw.api.webservice.exception.LoginException</code>.</p>	<ul style="list-style-type: none"> • <code>HttpServletRequest</code> object
Arbitrary user name / password pair	<code>login</code>	<p>Use the <code>login</code> method to pass an arbitrary user and password as <code>String</code> values and authenticate with ClaimCenter. For example, you might use a corporate implementation of single sign-on (SSO) authentication that stores information in HTTP headers other than the</p>	<ul style="list-style-type: none"> • <code>HttpServletRequest</code> object • Username as a <code>String</code> • Password as a <code>String</code>

Source of credentials	ServletUtils method name	Description	Method arguments
	getBasicAuthHeaders()	HTTP basic headers. You can get the user name and password and call this method. This method forces creation of a new session.	
	getAuthenticatedUser(HttpServletRequest req, boolean forceSession)	In a single sign-on environment, get the current session before calling login. Then, if necessary, do HTTP basic authentication.	
	checkAuthenticationResult()	Always check the return value. The method returns null if authentication failed.	
	login(String username, String password)	For login problems, this method might throw the exception gw.api.webservice.exception.LoginException.	

See also

- “Example of a servlet using multiple authentication types” on page 677

Example of a servlet using multiple authentication types

The following Gosu servlet runs in the context of the ClaimCenter application, and uses either Guidewire authentication or HTTP basic authentication. This example servlet responds to URL substrings that start with the string /test. If an incoming URL matches that pattern, the servlet displays information from properties of the response object.

The following code demonstrates this technique in the `service` method, which calls a separate method to do the main work of the servlet. Optionally, to check the HTTP request type, call the `getMethod` method on the servlet request object.

```
package mycompany.test

uses gw.servlet.Servlet
uses javax.servlet.http.HttpServlet
uses javax.servlet.http.HttpServletRequest
uses javax.servlet.http.HttpServletResponse
uses javax.servlet.http.HttpSession

@Servlet( \ path : String ->path.matches("/test(/.*)?"))
class TestingServlet extends HttpServlet {

    override function service(req: HttpServletRequest, response: HttpServletResponse) {
        //print("Beginning call to service()...")

        // SESSION AUTH : Get user from session if the client is already signed in.
        var user = gw.servlet.ServletUtils.getAuthenticatedUser(req, true)
        //print("Session user result = " + user?.displayName)

        // HTTP BASIC AUTH : If the session user cannot be authenticated, try HTTP Basic
        if (user == null) {
            try {
                user = gw.servlet.ServletUtils.getBasicAuthenticatedUser(req)
                //print("HTTP Basic user result = " + user?.displayName)
            } catch (e) {
                response.sendError(HttpServletRequest.SC_UNAUTHORIZED,
                    "Unauthorized. HTTP Basic authentication error.")
                return // Be sure to RETURN early because authentication failed!
            }
        }
        if (user == null) {
            response.sendError(HttpServletRequest.SC_UNAUTHORIZED,
                "Unauthorized. No valid user with session context or HTTP Basic.")
            return // Be sure to RETURN early because authentication failed!
        }
        // IMPORTANT: Execution reaches here only if a user succeeds with authentication.
        // Insert main servlet code here before end of function, which ends servlet request
        doMain(req, response, user )
    }

    // This method is called by our servlet AFTER successful authentication
    private function doMain(req: HttpServletRequest, response: HttpServletResponse, user : User) {
        assert(user != null)

        var responseText = "REQUEST SUCCEEDED\n" +
            "req.RequestURI: '${req.RequestURI}'\n" +
            "req.PathInfo: '${req.PathInfo}'\n" +
            "req.SessionID: ${req.SessionID}\n" +
            "user.DisplayName: ${user.DisplayName}\n"
    }
}
```

```
"req.RequestURL: '${req.RequestURL}'\n" +  
"authenticated user name: '${user.DisplayName}'\n"  
  
// Debugging message to the console  
//print(responseText)  
  
// For output response  
response.ContentType = "text/plain"  
response.setStatus(HttpServletRequest.SC_OK)  
response.getWriter.append(responseText)  
}
```

Test your servlet using multiple authentication types

You can use a servlet that uses either basic or Guidewire authentication to log in to ClaimCenter. Test this type of servlet by following this procedure.

Before you begin

In Studio, create a servlet that implements the `service` method, which calls the necessary `ServletUtils` methods. This topic uses the `mycompany.test.TestingServlet` servlet, as shown in “Example of a servlet using multiple authentication types” on page 677.

Procedure

1. Register this servlet in `servlets.xml`.
2. Run the QuickStart server at the command prompt:

```
gwb runServer
```

Do not log in to the ClaimCenter application.

3. Test the servlet with no authentication by going to the URL:

```
http://localhost:8080/cc/service/test
```

You see a message:

```
HTTP ERROR 401  
Problem accessing /cc/service/test. Reason:  
Unauthorized. No valid user with session context or HTTP Basic.
```

4. Log in to the ClaimCenter application with valid credentials.
5. Test the servlet with the ClaimCenter authenticated user by going to the URL:

```
http://localhost:8080/cc/service/test
```

You see a message:

```
REQUEST SUCCEEDED  
req.RequestURI: '/cc/service/test'  
req.PathInfo: '/test'  
req.RequestURL: 'http://localhost:8080/cc/service/test'  
authenticated user name: 'Super User'
```

6. To test the HTTP basic authentication:
 - a) Sign out of the ClaimCenter application to remove the session context.
 - b) Re-test your servlet, using a tool that supports adding HTTP headers for HTTP basic authentication.

Data extraction integration

ClaimCenter provides several mechanisms to generate messages, forms, and letters in custom text-based formats from ClaimCenter data. If an external system needs information from ClaimCenter about a claim, it can send requests to the ClaimCenter server by using the HTTP protocol.

Overview of using Gosu templates for data extraction

Incoming data extraction requests include what Gosu template to use and what information the request passes to the template. With this information, ClaimCenter searches for the requested data such as claim data, which is typically called the root object for the request. If you design your own templates, you can pass any number of parameters to the template. Next, ClaimCenter uses a requested template to extract and format the response. You can define your own text-based output format.

Possible output formats include the following formats.

- A plain text document with *name=value* pairs
- An XML document
- An HTML or XHTML document

You can fully customize the set of properties in the response and how to organize and export the output in the response. In most cases, you know your required export format in advance and it must contain dynamic data from the database. Templates can dynamically generate output as needed.

Gosu templates provide several advantages over a fixed, pre-defined format.

- With templates, you can specify the required output properties, so you can send as few properties as you want. With a fixed format, all properties on all subobjects might be required to support unknown use cases, so you must send all properties on all subobjects.
- Responses can match the native format of the calling system by customizing the template. This avoids additional code to parse and convert fixed-format data.
- Templates can generate HTML directly for custom web page views into the ClaimCenter database. Generate HTML within ClaimCenter or from linked external systems, such as intranet websites that query ClaimCenter and display the results in its own user interface.

The major techniques to extract data from ClaimCenter by using templates are described below.

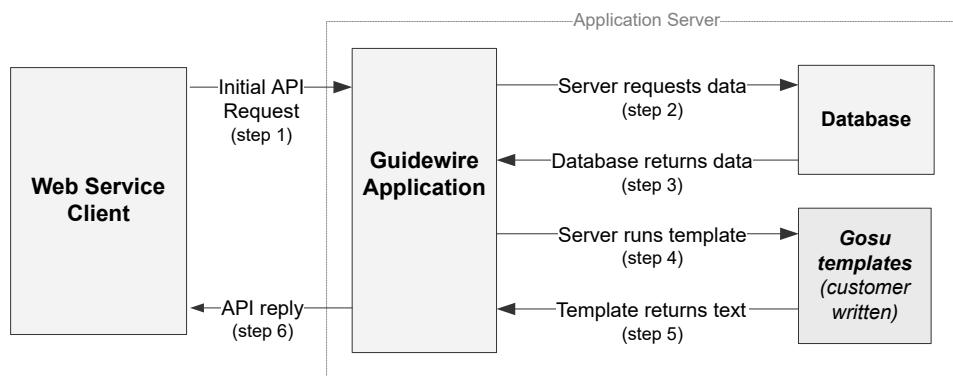
- For user interaction, write a custom servlet that uses templates. Create a custom servlet. A servlet generates HTML or other format pages for predefined URLs and you do not implement with PCF configuration. Your servlet implementation can use Gosu templates to extra data from the ClaimCenter database.

- For programmatic access, write a custom web service that uses templates. Write a custom web service. Custom web services support addressing each integration point in your network. Use the following design principles:
 - Write a different web service API for each integration point, rather than writing a single, general purpose web service API.
 - Name the methods in your web service APIs appropriately for each integration point. For example, if a method generates notification emails, name your method `getNotificationEmailText`.
 - Design your web service APIs to use method arguments with types that are specific to each integration point.
 - Use Gosu templates to implement data extraction. However, do not pass template data or anything with Gosu code directly to ClaimCenter for execution. Instead, store template data only on the server and pass only typesafe parameters to your web service APIs.

Data extraction using web services

You can write your own web services that use Gosu templates to generate results. The following diagram illustrates the data extraction process for a web service API that uses Gosu templates.

Web Service Data Extraction Flow



Every data extraction request includes a parameter indicating which Gosu template to use to format the response. You can add an unlimited number of templates to provide different responses for different types of root objects. For example, given a claim as a root object for a template, you could design different templates to export claim data, such as the following examples.

- A list of all notes on the claim
- A list of all open activities on the claim
- A summarized view of a claim

To provide programmatic access to ClaimCenter data, ClaimCenter uses Gosu, the basis of Guidewire Studio business rules. Gosu templates allow you to embed Gosu code that generates dynamic text from evaluating the code. Wrap your code with <% and %> characters for Gosu blocks and <%= and %> for Gosu expressions.

Once you know the root object, refer to properties on the root object or related objects just as you do as you write rules in Studio.

Before writing a template, decide which data you want to pass to the template.

For example, from a claim you could refer to many properties, including the following items.

`claim.LossDate`
Loss date

claim.LossType.Code

Code for a typelist value

claim.LossType.Name

Text description of the typelist value

claim.Policy.PolicyNumber

Policy number (different from its public ID)

The simplest Gosu expressions only extract data object properties, such as *claim.ClaimNumber*. For instance, suppose you want to export the claim number. You might use the following template, which despite its short length is a valid Gosu template in its entirety.

```
The number is <%= claim.claimNumber %>.
```

At run time, Gosu runs the code in the template block “`<%= ... %>`” and evaluates it dynamically.

```
The number is HO-1234556789.
```

Using Gosu templates for data extraction

Error handling in templates

By default, if Gosu cannot evaluate an expression because of an error or null value, it generates a detailed error page. It is best to check for blank or null values as necessary from Gosu so that you do not accidentally generate errors during template evaluation.

Getting parameters from URLs

If you want to get the value of URL parameters other than the root objects and/or check to see if they have a value use the syntax `parameters.get("paramnamehere")`. For instance, to check for the `xyz` parameter and export it, execute the following code statement.

```
<%= (parameters.get("xyz") == null)? "NO VALUE!" : parameters.get("xyz") %>
```

Example templates in the base configuration

The base configuration of ClaimCenter includes example Gosu templates for data extraction. Use Studio to refer to the files in the following folder: **configuration > config > templates > dataextraction**

Using loops in templates

Gosu templates can include loops that iterate over multiple objects such as every exposure or contact person associated with a claim. For example, suppose you want to display all claims associated with a policy. The corresponding template code with the root object `claims` might look something like the following code.

```
<% for (var thisClaim in claims) { %>
  Claim number: <%= thisClaim.ClaimNumber %>
<% } %>
```

This template might generate something like the following output.

```
Claim number: HO-123456:C0001
Claim number: HO-123456:C0002
Claim number: HO-123456:C0003
```

Structured export formats

HTML and XML are text-based formats, so there is no fundamental difference between designing a template for HTML or XML export compared to other plain text files. The only difference is that the text file must conform to HTML and XML specifications.

HTML results must be a well-formed HTML, ensuring that all properties contain no characters that might invalidate the HTML specification, such as unescaped “<” or “&” characters. This is particularly relevant for especially user-entered text such as descriptions and notes.

Systems that process XML often are very strict about syntax and well-formedness. Be careful not to generate text that might invalidate the XML or confuse the recipient. For instance, beware of unescaped “<” or “&” characters in a notes field. If possible, you could export data within an XML <CDATA> tag, which allows more types of characters and character strings without problems of unescaped characters.

Handling data model extensions in Gosu

If you added data model extensions, such as new properties within the `Claim` object, they are available work within Gosu templates just as in business rules within Guidewire Studio. For instance, just use expressions like `myClaim.myCustomField`.

Gosu template APIs for data extraction integration

Accessing Gosu libraries from within templates

You can access Gosu libraries from within templates similarly to how you access them from within Guidewire Studio by using the `Libraries` object.

```
<%= Libraries.Financials.getTotalIncurredForExposure(exposure) %>
```

Java classes called from Gosu in templates

Just like any other Gosu code, your Gosu code in your template use Java classes.

```
var myWidget = new mycompany.utils.Widget();
```

Logging in templates

Messages can be sent to a particular ClaimCenter log file by using the `gw.api.system.CCLoggerCategory` object. Retrieve the desired `Logger` object by calling the object's `forCategory` method and specifying the relevant log category. Text messages can be sent to the returned `Logger` object.

Typecode alias conversion in templates

As you write integration code in Gosu templates, you may also want to use ClaimCenter typelist mapping tools. ClaimCenter provides access to these tools by using the `$typecode` object.

```
typecode.getAliasByInternalCode("LossType", "ns1", claim.LossType)
```

This `$typecode` API works only within Gosu templates, not Gosu in general.

In addition, you can use the `TypecodeMapperUtil` Gosu utility class.

Inbound files integration

The base configuration of ClaimCenter includes a framework for configuring multiple integrations with external systems. This is achieved by processing file-based data. ClaimCenter provides the framework with a general processing mechanism and the `InboundFileHandler` interface which describes how to process data in the files.

You must provide configuration details and a class implementing the `InboundFileHandler` interface.

Supported file sources

Inbound files integration supports two types of file sources.

- Local - Files are read and archived on a directory in the local file system.
- Amazon S3 - Files are read and archived using an Amazon S3 bucket.

Inbound files integration process

Inbound files integration proceeds in the following stages.

1. Inbound files batch process:
 - a. Files are loaded from the input directory.
 - b. Inbound records are created in the database. Groups of records are split into chunks that are assigned the PENDING status.
Note: You can determine the number of files in chunks by setting the `ChunkSize` parameter in the integration configuration.
2. Work queue:
 - a. `InboundChunkWorkQueue` processes file records in chunks with the dedicated `FileHandler` implementation.

See also

- For information about configuring inbound files integration, see “Configure inbound files integration in the user interface” on page 687.
- For information about record statuses, see “Record statuses” on page 685.

Work queues

There are two work queue classes for processing inbound files.

- InboundFilePurgeWorkQueue
- InboundChunkWorkQueue

InboundFilePurgeWorkQueue

The work queue InboundFilePurgeWorkQueue checks if there are any inbound files for which the purge date has elapsed. If there are such files, the work queue removes them and associated records from the database.

The processing of a work item proceeds in the following steps.

1. The work queue removes all inbound records that belong to the inbound file from the database.
2. The work queue sets the foreign key value for the inbound file to null in InboundFilePurgeWorkItem.
3. The work queue removes the inbound file from the database.

InboundChunkWorkQueue

The work queue InboundChunkWorkQueue processes records within one chunk.

Implementing an integration

To implement an integration, create an implementation of the `InboundFileHandler` interface. Then, configure inbound files integration.

Create an InboundFileHandler implementation

About this task

Create your implementation of `InboundFileHandler`.

Procedure

1. Extend the `BaseInboundFileHandler` class.
2. Implement the following method.

```
@Override  
public void process(InboundRecord record, Bundle bundle, IntentionalLogger logger, Marker marker) {  
}
```

The `process` method takes the following parameters:

`record`

Inbound record

`bundle`

Bundle for processing any additional entities

`logger`

`IntentionalLogger` instance for logging events inside methods.

Note: For more information about intentional logging, see the *Administration Guide*.

`marker`

Marker for logging in `IntentionalLogger`

InboundFileHandler methods

By default, `BaseInboundFileHandler` implements the following `InboundFileHandler` methods. The methods can be overridden.

Table 1: InboundFileHandler methods

Method	Parameters	Description
boolean <code>isValid(String filename)</code>	<code>filename</code> – Full path name of the file.	Checks if a file is valid. If a file is not valid, it is marked as an error and the <code>OnLoadError</code> method of the handler is called.
boolean <code>shouldIgnore(String line, int lineNumber)</code>	<code>line</code> – Raw line as read from the file. <code>lineNumber</code> – Number of the line, starting from 1.	Checks whether to ignore a specific line in a file. For example, a header, a footer, or a comment.
boolean <code>isSubRecord(String line, int lineNumber)</code>	<code>line</code> – Raw line as read from the file. <code>lineNumber</code> – Number of the line, starting from 1.	Checks if a line in a file is a sub-record.
void <code>preProcess(IntentionalLogger logger, Marker marker)</code>	<code>logger</code> – IntentionalLogger instance for logging. <code>marker</code> – Marker instance for logging.	Performs pre-processing of the inbound files.
void <code>process(InboundRecord record, Bundle bundle, IntentionalLogger logger, Marker marker)</code>	<code>record</code> – Inbound record to process. <code>bundle</code> – Bundle for processing for any additional entities. <code>logger</code> – IntentionalLogger instance for logging. <code>marker</code> – Marker instance for logging.	Processes an inbound record and all sub-records associated with it.
void <code>postProcess(IntentionalLogger logger, Marker marker)</code>	<code>logger</code> – IntentionalLogger instance for logging. <code>marker</code> – Marker instance for logging.	Performs post-processing of the inbound files.
void <code>onLoadError(InboundFile file, String msg, IntentionalLogger logger, Marker marker)</code>	<code>file</code> – Inbound file that failed to load. <code>msg</code> – Error message. <code>logger</code> – IntentionalLogger instance for logging. <code>marker</code> – Marker instance for logging.	Performs additional error handling for when <code>InboundFileHandler</code> encounters an exception during processing of an inbound file.
void <code>onProcessError(InboundRecord record, String msg, IntentionalLogger logger, Marker marker)</code>	<code>record</code> – Inbound record that failed to process. <code>msg</code> – Error message. <code>logger</code> – IntentionalLogger instance for logging. <code>marker</code> – Marker instance for logging.	Performs additional error handling for when <code>InboundFileHandler</code> encounters an exception during processing of an inbound record.

Record statuses

An inbound record can be assigned the following statuses.

- PENDING – The record has not been processed yet.

- PROCESSED – The record was processed successfully.
- ERROR – The record could not be processed.
- IGNORE – The record will be ignored during the processing.
- SKIPPED – The record was skipped manually.

An inbound chunk can be assigned the following statuses.

- PENDING – Records are ready to be processed.
- PROCESSED – All records in the chunk were processed successfully.
- PROCESSED_WITH_ERRORS – There was an error during processing of records in the chunk.

Configuring inbound files integration

You can specify inbound configurations in the ClaimCenter interface or use the `InboundConfigPlugin`.

Configuration parameters for local integrations

Set the following parameters for local integrations:

- Name – Name of a property. For example: `payment-files`. This value must be unique.
- Input directory – Directory with input files. For example: `tmp/in-files`. This value must be unique.
- Archive directory – Directory where archive files will be stored after processing. For example: `tmp/out-files`.
- Chunk size – Parameter specifying if records in the file will be processed in parallel or sequentially. You can set the following values.

Value	Effect
0	All records in the file will be processed in a sequence by one work queue.
1 - ...	Records in the file will be processed in parallel by several work queues. The value specifies the number of records processed by each work queue.

Note: Each record can have several sub-records. Sub-records are not split between chunks.

- File handler class – Fully qualified name of a class containing the inbound file handler logic. For example: `com.guidewire.InboundPaymentFileHandler`. The class must extend `BaseInboundFileHandler`. On start-up, during uploading of the sources, this class is validated to check whether it implements `InboundFileHandler`.
- Days till purge – Number of days before the processed inbound files and records are purged from the database. The value must be between 9 and 730. The default value is 365.

Configuration parameters for remote Amazon S3 integrations

Set the following parameters for remote integrations.

- Name – Name of a property. For example: `payment-files`. This value must be unique.
- Amazon user profile name – Name of the Amazon S3 profile that will be used for authentication.

Note: The user key and secret key of the Amazon S3 profile must be stored in the `~/.aws/credentials` file, under the profile name you provide in the configuration.

Alternatively, you can authenticate with an EC2 role.

- Input bucket – Name of the Amazon S3 bucket with the input files.
- Input prefix (optional) – Full Amazon S3 bucket path where the input files are located.
- Archive bucket – Name of the Amazon S3 bucket where the files will be archived.
- Archive prefix (optional) – Full Amazon S3 bucket path where the files will be archived.

- Chunk size – Parameter specifying if records in the file will be processed in parallel or sequentially. You can set the following values.

Value	Effect
0	All records in the file will be processed in a sequence by one work queue.
1 - ...	Records in the file will be processed in parallel by several work queues. The value specifies the number of records processed by each work queue.

Note: Each record can have several sub-records. Sub-records are not split between chunks.

- File handler class – Fully qualified name of a class containing the inbound file handler logic. For example: `com.guidewire.InboundPaymentFileHandler`. The class must extend `BaseInboundFileHandler`. On startup, during uploading of the sources, this class is validated to check whether it implements `InboundFileHandler`.
- Days till purge – Number of days before the processed inbound files and records are purged from the database. The value must be between 9 and 730. The default value is 365.

Configuring the Amazon S3 endpoint

To use inbound files integration with Amazon S3, you must set the AWS S3 endpoint to one that matches the region of the S3 bucket you are using.

To set the endpoint, edit the value of the `AWS_S3_ENDPOINT` runtime property in the configuration group.

For example:

```
AWS_S3_ENDPOINT="s3.eu-central-1.amazonaws.com"
```

For information about runtime properties, see “Runtime properties” in the *Administration Guide*.

Configure inbound files integration in the user interface

About this task

Add a configuration in the ClaimCenter interface.

Procedure

1. Navigate to **Administration > Utilities > Inbound Files**.
2. Click **Inbound File Configs**.
3. Depending on the integration type, select **Local Storage Configs** or **Amazon S3 Storage Configs**.
4. Click **Add**.
5. Specify the configuration parameters and click **Update**.

Configure inbound files integration with InboundConfigPlugin

You can add or edit a configuration of inbound files integration with `InboundConfigPlugin`. This plugin can load all configurations from the `config/inbound/InboundFileConfiguration.xml` file or from an external file specified during server startup.

Plugin InboundConfigPlugin

`InboundConfigPlugin` is located in the `InboundConfigPlugin.gwp` file.

```
<?xml version="1.0"?>
<plugin
    interface="IStartablePlugin"
    name="InboundConfigPlugin">
```

```

<plugin-java
    javaClass="com.guidewire.inboundfile.plugin.InboundConfigPlugin">
    <param
        name="externalConfigFileLocation"
        value="/tmp/guidewire/inbound/config/InboundFileConfiguration.xml"/>
    <param
        name="updateExisting"
        value="false"/>
</plugin-java>
</plugin>

```

Where:

- **externalConfigFileLocation** – Valid path to the external file with configuration definitions.
- **updateExisting** – Parameter specifying whether existing configurations in the database must be updated.

[External configuration file](#)

The external configuration file must be based on an .xsd definition file. The following code is an example of a .xsd definition file.

```

<?xml version="1.0" encoding="utf-8"?>
<xs:schema targetNamespace="http://guidewire.com/inbound" xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns="http://guidewire.com/inbound" elementFormDefault="qualified">
    <xs:element name="InboundConfigurations">
        <xs:complexType>
            <xs:sequence>
                <xs:element type="InboundLocalConfiguration" name="LocalConfiguration" minOccurs="0" maxOccurs="unbounded" />
                <xs:element type="InboundS3Configuration" name="S3Configuration" minOccurs="0" maxOccurs="unbounded" />
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:complexType name="InboundConfigurationBase">
        <xs:sequence>
            <xs:element type="xs:string" name="Name"/>
            <xs:element type="xs:string" name="Prefix"/>
            <xs:element type="xs:string" name="Extension"/>
            <xs:element type="xs:string" name="FileHandlerClass"/>
            <xs:element type="xs:integer" name="DaysTillPurge"/>
            <xs:element type="xs:string" name="TemporaryDirectory"/>
        </xs:sequence>
        <xs:attribute name="env" type="xs:string"/>
    </xs:complexType>
    <xs:complexType name="InboundLocalConfiguration">
        <xs:complexContent>
            <xs:extension base="InboundConfigurationBase">
                <xs:sequence>
                    <xs:element type="xs:string" name="PermanentDirectory"/>
                </xs:sequence>
            </xs:extension>
        </xs:complexContent>
    </xs:complexType>
    <xs:complexType name="InboundS3Configuration">
        <xs:complexContent>
            <xs:extension base="InboundConfigurationBase">
                <xs:sequence>
                    <xs:element type="xs:string" name="ProfileName" minOccurs="0"/>
                    <xs:element type="xs:string" name="PermanentBucketName"/>
                    <xs:element type="xs:string" name="PermanentPrefix"/>
                </xs:sequence>
            </xs:extension>
        </xs:complexContent>
    </xs:complexType>
</xs:schema>

```

[Loading pre-defined inbound configurations](#)

You can load pre-defined inbound configurations during server startup. All configurations that you specify are mapped onto their corresponding entities.

Add pre-defined inbound configurations to the `InboundFileConfiguration.xml` file.

Example configuration:

```

<?xml version="1.0"?>
<InboundConfigurations
    xmlns="http://guidewire.com/inbound">

```

```
<LocalConfiguration env="DEV">
  <Name>ConfigOne</Name>
  <FileHandlerClass>gw.api.inboundfile.ExampleInboundFileHandler</FileHandlerClass>
  <ChunkSize>0</ChunkSize>
  <DaysTillPurge>5</DaysTillPurge>
  <InputDirectory>inputdir</InputDirectory>
  <ArchiveDirectory>archivedir</ArchiveDirectory>
</LocalConfiguration>
<LocalConfiguration env="PROD">
  <Name>ConfigOne</Name>
  <FileHandlerClass>gw.api.inboundfile.RealInboundFileHandler</FileHandlerClass>
  <ChunkSize>0</ChunkSize>
  <DaysTillPurge>5</DaysTillPurge>
  <InputDirectory>prodinputdir</InputDirectory>
  <ArchiveDirectory>prodarchivedir</ArchiveDirectory>
</LocalConfiguration>
<S3Configuration>
  <Name>ConfigTwo</Name>
  <FileHandlerClass>gw.api.inboundfile.ExampleInboundFileHandlerTwo</FileHandlerClass>
  <ChunkSize>0</ChunkSize>
  <DaysTillPurge>5</DaysTillPurge>
  <ProfileName>guidewire</ProfileName>
  <InputBucketName>inputbucket</InputBucketName>
  <InputPrefix>input</InputPrefix>
  <ArchiveBucketName>archivebucket</ArchiveBucketName>
  <ArchivePrefix>archive</ArchivePrefix>
</S3Configuration>
</InboundConfigurations>
```

Environment-specific configuration

To load configuration for a specific environment, use the `env` attribute to specify the environment.

Example configuration with the `env` attribute:

```
<LocalConfiguration env="DEV">
  <!-- loaded on dev environment only -->
</LocalConfiguration>
<LocalConfiguration env="PROD">
  <!-- loaded on prod environment only -->
</LocalConfiguration><LocalConfiguration>
  <!-- loaded on all environments -->
</LocalConfiguration>
```


Outbound files integration

The base configuration of ClaimCenter includes a framework that supports creating files for external systems. The files are created from records in the database. ClaimCenter provides the framework with a general processing mechanism and the `OutboundFileHandler` interface which describes how to process data in the records.

You must provide configuration details and a class implementing the `OutboundFileHandler` interface.

Supported file destinations

Outbound files integration supports two types of external file outputs:

- Local - Files are transferred to a directory in the local file system.
- Amazon S3 - Files are transferred to an Amazon S3 bucket.

Outbound files integration process

Outbound files integration proceeds in the following stages.

1. Outbound files batch process:
 - a. Outbound records are loaded from the database.
 - b. Outbound files are created and saved in the specified location.
2. Work queue:
 - a. The work queue writes file records to the specified outbound file with the dedicated `OutboundFileHandler` implementation.

Work queues

There are two work queue classes for processing outbound files.

- `OutboundFilePurgeWorkQueue`
- `OutboundRecordPurgeWorkQueue`

The work queues are enabled by default and listed in the `work-queue.xml` file

```
<work-queue
    workQueueClass="com.guidewire.outboundfile.workqueue.OutboundRecordPurgeWorkQueue"
    progressInterval="60000">
    <worker
        instances="1"/>
</work-queue>
<work-queue>
```

```

workQueueClass="com.guidewire.outboundfile.workqueue.OutboundFilePurgeWorkQueue"
progressInterval="60000"
<worker
  instances="1"/>
</work-queue>

```

OutboundFilePurgeWorkQueue

The work queue OutboundFilePurgeWorkQueue checks if there are any outbound files for which the purge date has elapsed. If there are such files, the work queue removes them and associated records from the database.

The processing of a work item proceeds in the following steps.

1. The work queue removes all outbound records that belong to the outbound file from the database.
2. The work queue sets the foreign key value for the outbound file to null in OutboundFilePurgeWorkItem.
3. The work queue removes the outbound file from the database.

OutboundRecordPurgeWorkQueue

The work queue OutboundRecordPurgeWorkQueue checks if there are any outbound records for which the purge date has elapsed and that have the SKIPPED status. If there are such records, the work queue removes them from the database.

The processing of a work item proceeds in the following steps.

1. The work queue sets the foreign key value for the outbound record to null in OutboundRecordPurgeWorkItem.
2. The work queue removes the outbound record from the database.

Implementing an integration

To implement an integration, create an implementation of the OutboundFileHandler interface. Then, configure outbound files integration.

Create an OutboundFileHandler implementation

About this task

Create your implementation of OutboundFileHandler.

Procedure

1. Extend the BaseOutboundFileHandler class.
2. Implement the following methods.

```

void open(String tempFilename, IntentionalLogger logger, Marker marker);
void process(OutboundRecord record, IntentionalLogger logger, Marker marker) throws
OutboundFileProcessingException;

```

Where:

- **tempFilename** – Name of the outbound file where the records are written.
- **record** – Outbound record.
- **logger** – IntentionalLogger instance for logging events inside methods.

Note: For more information about intentional logging, see *Administration Guide*.

- **marker** – Marker for logging in IntentionalLogger.
- **OutboundFileProcessingException** – Exception thrown when there is an error during processing records.

OutboundFileHandler methods

By default, `BaseOutboundFileHandler` implements the following `OutboundFileHandler` methods. The methods can be overridden.

Table 2: OutboundFileHandler methods

Method	Parameters	Description
<code>public void open(String tempFilename, IntentionalLogger logger, Marker marker)</code>	<code>tempFilename</code> – Full path name of the file. <code>logger</code> – <code>IntentionalLogger</code> instance for logging. <code>marker</code> – <code>Marker</code> instance for logging.	Creates and opens a file.
<code>public void process(OutboundRecord record, IntentionalLogger logger, Marker marker)</code>	<code>record</code> – Outbound record. <code>logger</code> – <code>IntentionalLogger</code> instance for logging. <code>marker</code> – <code>Marker</code> instance for logging.	Writes a record to the open file.

Record statuses

An outbound record can be assigned the following statuses.

- PENDING – The record has not been processed yet.
- PROCESSED – The record was processed successfully.
- ERROR – The record could not be processed.
- SKIPPED – The record was skipped manually.

Configure outbound files integration

You can specify outbound configurations in the ClaimCenter interface or use the `OutboundConfigPlugin`.

Configuration parameters for local configurations

Set the following parameters for local configurations:

- Name – Name of the outbound file destination.
- Temporary directory – Path to the temporary directory for the output file.
- Permanent directory – Path to the permanent directory for the output file.
- Prefix – Prefix for the batch identifier and the output file.
- Extension – Extension of the output file.
- File handler class – Fully qualified name of a class containing the outbound file handler logic. For example: `com.guidewire.BaseOutboundFileHandler`. The class must extend the `OutboundFileHandler` interface.
- Days till purge – Number of days before the processed outbound files and record are purged from the database. The value must be between 9 and 730. The default value is 365.

Configuration parameters for remote Amazon S3 configurations

Set the following parameters for remote configurations:

- Name – Name of the outbound file destination.
- Amazon user profile name – Name of the Amazon S3 profile that will be used for authentication.

Note: The user key and secret key of the Amazon S3 profile must be stored in the `~/.aws/credentials` file, under the profile name you provide in the configuration.

Alternatively, you can authenticate with an EC2 role.

- Temporary directory – Path to the temporary directory for the output file.
- Permanent directory bucket – Name of the Amazon S3 bucket for the outbound files.
- Permanent directory prefix – (optional) Full Amazon S3 bucket path where the outbound files will be located.
- Prefix – Prefix for the batch identifier and the output file.
- Extension – Extension of the output file.
- File handler class – Fully qualified name of a class containing the outbound file handler logic. For example: `com.guidewire.BaseOutboundFileHandler`. The class must extend the `OutboundFileHandler` interface.
- Days till purge – Number of days before the processed outbound files and records are purged from the database. The value must be between 9 and 730. The default value is 365.

Configuring the Amazon S3 endpoint

To use outbound files integration with Amazon S3, you must set the AWS S3 endpoint to one that matches the region of the S3 bucket you are using.

To set the endpoint, edit the value of the `AWS_S3_ENDPOINT` runtime property in the configuration group.

For example:

```
AWS_S3_ENDPOINT="s3.eu-central-1.amazonaws.com"
```

For information about runtime properties, see “Runtime properties” in the *Administration Guide*.

Configure outbound files integration in the user interface

About this task

Add a configuration in the ClaimCenter interface.

Procedure

1. Click **Administration > Utilities > Outbound Files**.
2. Click **Outbound File Configs**.
3. Depending on the integration type, select **Local Storage Configs** or **Amazon S3 Storage Configs**.
4. Click **Add**.
5. Specify the configuration parameters and click **Update**.

Configure outbound files integration with OutboundConfigPlugin

Add or edit a configuration with `OutboundConfigPlugin`. This plugin can load all configurations from the `config/outbound/OutboundFileConfiguration.xml` file or from an external file specified during server startup.

The `OutboundConfigPlugin` plugin

`OutboundConfigPlugin` and all its parameters are in the `OutboundConfigPlugin.gwp` file.

```
<?xml version="1.0"?>
<plugin
  interface="IStartablePlugin"
  name="OutboundConfigPlugin">
  <plugin-xml>
    <param
      name="externalConfigFileLocation"
      value="/tmp/guidewire/outbound/config/OutboundFileConfiguration.xml"/>
    <param
      name="updateExisting"
      value="false"/>
```

```
</plugin>
</plugin>
```

Where:

- **externalConfigFileLocation** – Valid path to the external file with configuration definitions.
- **updateExisting** – Parameter specifying whether existing configurations in the database must be updated.

External configuration file

The external configuration file must be based on an .xsd definition file.

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema targetNamespace="http://guidewire.com/outbound" xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns="http://guidewire.com/outbound" elementFormDefault="qualified">
  <xs:element name="OutboundConfigurations">
    <xs:complexType>
      <xs:sequence>
        <xs:element type="OutboundLocalConfiguration" name="LocalConfiguration" minOccurs="0" maxOccurs="unbounded" />
        <xs:element type="OutboundS3Configuration" name="S3Configuration" minOccurs="0" maxOccurs="unbounded" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:complexType name="OutboundConfigurationBase">
    <xs:sequence>
      <xs:element type="xs:string" name="Name"/>
      <xs:element type="xs:string" name="Prefix"/>
      <xs:element type="xs:string" name="Extension"/>
      <xs:element type="xs:string" name="FileHandlerClass"/>
      <xs:element type="xs:integer" name="DaysTillPurge"/>
      <xs:element type="xs:string" name="TemporaryDirectory"/>
    </xs:sequence>
    <xs:attribute name="env" type="xs:string"/>
  </xs:complexType>
  <xs:complexType name="OutboundLocalConfiguration">
    <xs:complexContent>
      <xs:extension base="OutboundConfigurationBase">
        <xs:sequence>
          <xs:element type="xs:string" name="PermanentDirectory"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
  <xs:complexType name="OutboundS3Configuration">
    <xs:complexContent>
      <xs:extension base="OutboundConfigurationBase">
        <xs:sequence>
          <xs:element type="xs:string" name="ProfileName" minOccurs="0"/>
          <xs:element type="xs:string" name="PermanentBucketName"/>
          <xs:element type="xs:string" name="PermanentPrefix"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:schema>
```

Environment-specific configuration

To load configuration for a specific environment, in the .xsd definition file, use the **env** attribute to specify the environment.

Example configuration with the **env** attribute:

```
<LocalConfiguration env="DEV"> ...
  <!-- loaded on dev environment only -->
</LocalConfiguration>
<LocalConfiguration env="PROD">
  <!-- loaded on prod environment only -->
</LocalConfiguration>
<LocalConfiguration>
  <!-- loaded on all environments -->
</LocalConfiguration>
```


Custom processing by using work queues and batch processes

ClaimCenter supports two modes of processing:

- A work queue that operates on a batch of items in parallel.
- A batch process that operates on a batch of items sequentially.

Custom processing

It is possible to implement custom processing to supplement the standard processes that Guidewire provides in the base configuration.

Overview of custom processes

Processing modes

ClaimCenter supports two modes of processing:

- Work queue
- Batch process

Work queue

A work queue operates on a batch of items in parallel. ClaimCenter distributes work queues across all servers in a ClaimCenter cluster that have the appropriate role. In the base configuration, Guidewire assigns this functionality to the `workqueue` server role.

A work queue comprises the following components:

- A processing thread, known as a *writer*, that selects a group (batch) of business records to process. For each business record (a claim record, for example), the writer creates an associated work item.
- A queue of selected work items.
- One or more tasks, known as *workers*, that process the individual work items to completion. Each worker is a short-lived task that exists in a thread pool. Each work queue on a cluster member shares the same thread pool. By default, each work queue starts a single worker on each server with the appropriate role, unless configured otherwise.

Work queues are suitable for high volume batch processing that requires the parallel processing of items to achieve an acceptable throughput rate.

Batch process

A batch process operates on a batch of items sequentially. Batch processes are suitable for low volume batch processing that achieves an acceptable throughput rate as the batch process processes items in sequence. For example, writers for work queues operate as batch processes because they can select items for a batch and write them to their work queues relatively quickly.

Choosing a mode for a custom process

Consider the following factors to decide between developing a custom work queue and developing a custom batch process:

- The volume of items in a typical batch
- The duration of the batch processing window

For business oriented batch processing, such as invoicing or ageing, Guidewire recommends that you develop a custom work queue. Work queues process items in parallel tasks distributed across all servers in a ClaimCenter cluster that have the `workqueue` server role.

IMPORTANT: Regardless of the volume of items or the duration of the processing window, Guidewire strongly recommends that you implement all custom batch processing as a custom work queue.

About scheduling ClaimCenter processes

It is possible to schedule many, if not most, of the ClaimCenter processes for execution at periodic intervals. Guidewire recommends that you take the following information into account in setting up a processing schedule.

"Nightly Batch Processing" Processes business transactions accumulated at the end of specific business periods, such as business days, months, quarters, or years.

"Daytime Batch Processing" Defers to periodic asynchronous background processing complex transactional processing triggered by user actions.

Depending on your custom process, certain implementation details can vary.

The scheduler configuration file

File `scheduler-config.xml` defines the default schedule that ClaimCenter uses to launch many of the processes, including writer processes for work queues. You can find this file in the following location in the Studio **Project** window:

configuration > config > scheduler

It is possible to define your own schedule for a process. See the *Administration Guide* for more information.

Night-time processing

Night-time processing typically processes relatively large batches of work, such as processing premium payments that accumulate during the business day. Each type of night-time processing runs once during the night, when users are not active in the application.

Because most kinds of night-time processing typically operate on large batches and have rigid processing windows, develop your night-time process as a custom work queue. Night-time processing often requires processing items in parallel to achieve sufficient throughput. Large workloads of nightly processing are typically too severe for the servers in a ClaimCenter cluster that have the `batch` server role.

For night-time batch processing, you typically configure a custom work queue table to segregate the work items of different work queues from each other. Each type of night-time processing typically has many worker tasks simultaneously accessing the queue for available work items. Using the same work queue table for all types of nightly work queues can degrade processing throughput due to table contention. In addition, segregating work items from different work queues into separate tables can ease recovery of failed work items before the nightly processing window closes.

Night-time processing frequently requires chaining so that completion of one type of batch processing starts another type of follow-on processing. Regardless of implementation mode – work queue or batch process – you most likely must develop custom process completion logic for your type of night-time processing.

Daytime processing

Typically, daytime processes are relatively small batches of work, such as reassigning activities escalated by users. Daytime processes run frequently during the day, while users are active in the application. You might schedule different types of daytime processes to run every hour or even every few minutes.

Even though daytime processing typically operates on small batches and lacks rigid processing windows, develop your type of daytime batch processing as a custom work queue. Although daytime processing often achieves sufficient throughput by processing items sequentially, its workload on the servers with the `batch` server role can slow overall performance of the application. As a work queue, worker tasks on multiple servers (with the `workqueue` server role), can perform the work in parallel.

If you develop your daytime processing as a custom work queue, you typically can use the standard work queue table for your work items. Different types of daytime processing run intermittently and in different bursts of relatively short work. So, table contention between various types of daytime batch processing often is minimal.

Daytime processes seldom requires chaining. Completion of one type of daytime process seldom starts another type of follow-on processing. Regardless of implementation mode, you most likely do not need to develop custom process completion logic for your type of daytime processing.

About manually executing ClaimCenter processes

ClaimCenter provides the following Server Tools screens for use by administrators in manually executing the different application processes.

Server Tools screen	More information
Batch Process Info	Administration Guide
Work Queue Info	Administration Guide

Batch processing typecodes

ClaimCenter identifies and manages application batch processing by typecodes in the `BatchProcessType` typelist. Each type of batch processing, whether implemented as a batch process or a work queue, has a unique typecode in the typelist. Whenever you develop a type of custom batch processing, begin by defining a typecode for it the `BatchProcessType` typelist.

You use the typecode for your type of custom batch processing in the following ways:

- Arguments to some of the methods in your custom work queue or batch process class
- An argument to the maintenance tools command and web service that enable you to start a batch processing run
- Categorizing how your custom batch process can be run, from the administrative user interface, on a schedule, or from the maintenance tools command prompt or web service
- A case clause in the Batch Processing Completion plugin for your type of batch processing

Regardless the mode of batch processing you implement, first define a new typecode in the `BatchProcessType` typelist for your type of batch processing..

See also

- “Define a typecode for a custom work queue” on page 706

Custom work queues

A work queue is code that runs without human intervention as a background process on multiple servers to process the units of work for a batch in parallel. Develop a custom work queue if you require the processing of units of work in parallel to achieve an acceptable throughput rate.

About custom work queues

A custom work queue comprises the following components.

Writer

The `findTargets` method on a Gosu class that extends the `WorkQueueBase` class that selects the units of work for a batch and writes work items for them in the work queue table.

Work queue

An entity type that implements the `WorkItem` delegate, such as the `StandardWorkItem` entity, to establish the database table for the work queue and its work items.

Worker

The `processWorkItem` method on the same Gosu class that extends the `WorkQueueBase` base class that processes the units of work identified on the work queue by work items

Starting the writer initiates a run of the type of batch processing that a work queue performs. The batch is complete when the workers exhaust the queue of all work items in the batch, except those they fail to process successfully.

About custom work queue classes

You implement the code for your writer and your workers as methods on a single Gosu class. You must derive your custom work queue class from the `WorkQueueBase` class. This base class and the work queue framework provide most of the logic for managing the work items in your work queue. You typically need to override only a few methods in the base class to implement the code for your writer and your workers.

Work queue writer

You implement the writer for your work queue by overriding the `findTargets` method inherited from the base class. Your code selects the units of work for a batch and returns an iterator for the result set. The work queue framework then uses the iterator to create and insert work items into your work queue. Each work item holds the instance ID of a unit of work in your result set.

For example, your custom work queue sends email about overdue activities to their assignees. In this example, instances of the `Activity` entity type are the units of work for the work queue. In your `findTargets` methods, you query the database for activity instances in which the last viewed date of an open activity exceeds 30 days. You return an iterator to the result set, and then the framework creates a work item for each element in the result.

Work queue workers

You implement the workers of your work queue by overriding the `processWorkItem` method inherited from the base class. The work queue framework calls the method with a work item as the sole parameter. Your code accesses the unit of work identified in the work item and processes it to completion. Upon completion, your code returns from the method, and the work queue framework deletes the work item from the work queue.

For example, the units of work for your work queue are open activities that have not been viewed in the past 30 days. In your `processWorkItem` method, you access the activity instance identified by the work item. Then, you generate and send an email message to the assignee of that activity. After your code sends the email, it returns from the method. The framework then deletes the work item and updates the count of successfully processed work items in the process history for the batch.

Work queues and work item entity types

A work item is an instance of entity type that implements the `WorkItem` delegate, such as `StandardWorkItem`. The entity type provides the database table in which the work items for your custom work queue persist. Work items in the work queue table are accessible from all servers in a ClaimCenter cluster. Thus, workers can access the work items asynchronously, in parallel, and distribute the work items to servers with the `workqueue` server role.

Work items have many properties that the work queue framework uses to manage work items and the process histories of work queue batches. If you use `StandardWorkItem` for your work queue, the only field on a work item that your custom code uses is the `Target` field. The `Target` field holds the ID of a unit of work that your writer selected. The

code for your writer and your workers can safely ignore the other properties on work item instances. However, you can define a custom work item type with additional fields for your custom code to use.

Custom work item types

Guidewire recommends that you define your custom work item entities as `keyable`, rather than `retireable`. If you define a custom work item as `retireable`, then you also need to create a custom batch process to purge the work items after ClaimCenter processes them and they become old and stale. This batch process also needs to clean up failed work items as well. Any delay in purging failed or stale work items can prevent further operations such as archiving.

Work queues that use StandardWorkItem

Guidewire supports creating multiple work queues that use the same work item type only if that work item is the `StandardWorkItem` type.

In the ClaimCenter data model, the `StandardWorkItem` entity type implements the `WorkItem` delegate. The `StandardWorkItem` entity type thus inherits the following from the `WorkItem` delegate:

- Methods to manipulate the work item
- Entity fields that store such items as `Status`, `Priority`, `Attempts`, and other similar types of information

Guidewire marks the `StandardWorkItem` entity type as `final`. Thus, it is not possible to subtype this entity type.

The `StandardWorkItem` entity type contains a `QueueType` typekey, which obtains its value from the `BatchProcessType` typelist. It is the use of this typekey that makes it possible for multiple work queues to use the `StandardWorkItem` type simultaneously.

For example, to query for all `StandardWorkItem` work items of a certain type, use a query that is similar to the following:

```
var target = Query.make(StandardWorkItem).compare(StandardWorkItem#QueueType, Equals, BatchProcessType.TC_CUSTOMWORKITEM)
```

This query returns all work items for the process that match the filter criteria. The query includes failed work items and work items in progress, not just the work items available for selection.

Error: Two work queues cannot use the same work item type

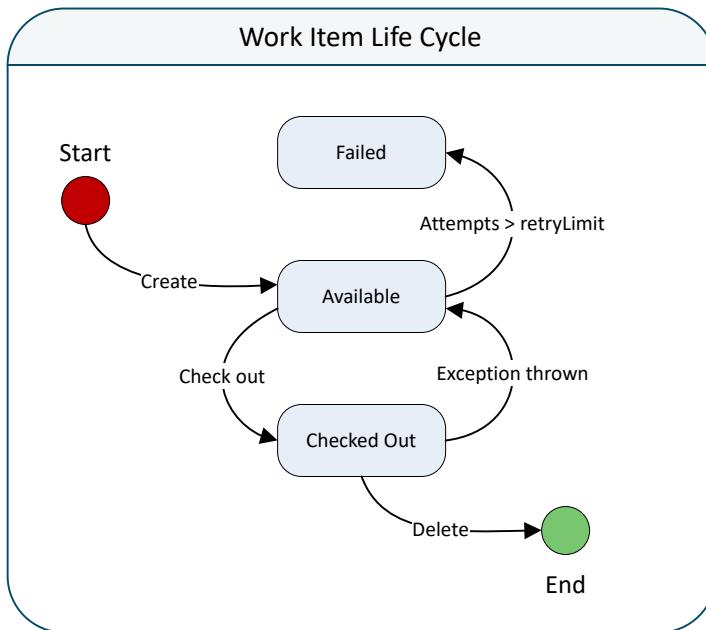
Guidewire does not support creating multiple work queues that use the same work item type, except for the `StandardWorkItem` type. If you attempt to do so using a work item type other than `StandardWorkItem`, ClaimCenter throws the following error:

```
Two work queues cannot use the same work item type at runtime: workItemType.Name
```

It is not possible for workers from different work queues to process work items from the same work item database table. In this circumstance, it is not possible for the different sets of workers to determine which work items to process. In any case, it is very likely that there can be database contention issues along with race conditions between the workers in picking up the work items from the queue.

Lifecycle of a work item

The `Status` field of a work item records the current state of its lifecycle. The work queue framework manages this field and the lifecycle of work items for your writer and workers. The following diagram illustrates the lifecycle.



A work item begins life after the writer selects the units of work for a batch and returns an iterator for the collection. The framework then creates work items that reference the units of work for a batch. The initial state of a work item is **available**. At the time the framework checks out a quota of work items for a worker, the state of the work items becomes **checkedout**. The framework then hands the quota of work items to the worker one at a time. After a worker finishes processing a work item successfully, the framework deletes it from the work queue and updates the statistics in the process history for the batch.

Returning work items to the work queue

Sometimes a worker cannot finish processing a work item successfully. For example, a network resource may be temporarily unavailable. Or, a concurrent data change exception (CDCE) can occur if multiple workers simultaneously update common entity data in the domain graphs of two separate units of work. Whenever errors occur, the worker throws an exception to return the work item to the work queue as **available** again.

Whenever a worker returns a work item to the work queue, ClaimCenter increments the **Attempts** property on the work item. If the value of **Attempts** remains below the retry limit for the worker, the state of the work item remains **available**. The work item is subsequently checked out in a quota for the same or another worker. If the temporary error condition clears, the next worker completes it successfully and ClaimCenter removes the work item from the work queue.

Work items that fail

Whenever a worker returns a checked out work item and the **Attempts** property exceeds the work item retry limit, the state of the work item becomes **failed**. Failed work items end their lifecycle in this state. Workers ignore items with a status of **failed** and no longer attempt to process them. Someone must determine the reason for the failure of a work item and take corrective action. They can then remove the failed work item from the work queue.

See also

- For more information on the lifecycle of work queues, see the *Administration Guide*.

Considerations for developing a custom work queue

Before you develop the Gosu code for your custom work queue, you need to decide which of the following you want to use for your work queue table:

- A `StandardWorkItem` entity type

- A custom work item type

The following reasons outline why you would want to define a custom work item type:

- Include custom fields on work items not available on the `StandardWorkItem` entity type
- Avoid table contention with other work queues that typically process large batches at the same time

Whenever you define a custom work item type, you must implement the `createWorkItem` method that your custom work queue inherits from `WorkQueueBase`.

If you create a custom work queue, your writer can pass more fields to the workers than a target field. On `StandardWorkItem`, the `Target` field is an object reference to an entity instance, which represents the unit of work for the workers. In a custom work item, you can define as many fields as you want to pass to the workers. Your custom work item itself can be the unit of work.

Guidewire recommends that you use custom work queue types for work queues with typically large batches that potentially run at the same time as other work queues with large batches. Especially if developing a custom work queue for night-time processing, consider using a custom work queue subtype instead of using `StandardWorkItem`. If you create a custom work item subtype to avoid table contention, you often define a single field for the writer to set, much like the `Target` field on `StandardWorkItem`. By convention, you name the single field with the same name as the entity type, not the generic name `Target`.

IMPORTANT: Do not create multiple work queues that use the same work item type other than the `StandardWorkItem` type. If you attempt to do so, ClaimCenter throws an error.

Linking custom work items to target entities

In creating a custom work item, you need to create a link from the custom work item table to the entity table that is the target of the work item. To create this link, add a `column` element to your custom work item and use the following parameters.

Name	Value
name	Target
type	softentityreference
nullok	false

For `Target`, enter the name of the entity on which the work items acts. Use `softentityreference` (a soft link) rather than `foreignkey` (a hard link). A `softentityreferece` is a foreign key for which ClaimCenter does not enforce integrity constraints in the database. The use of a hard foreign key in this context would do the following:

- Require that you delete the work item row from the database before you delete or archive the linked entity row.
- Can force ClaimCenter to include the custom work item table in the main domain graph. It is not usually desirable to include such administrative entities in the domain graph for archiving along with the other data.

Keyable work item entities

Guidewire recommends that you define your custom work item entities as `keyable`, rather than `retireable`. If you define a custom work item as `retireable`, then you also need to create a custom batch process to purge the work items after ClaimCenter processes them and they become old and stale. This batch process also needs to clean up failed work items as well. Any delay in purging failed or stale work items can prevent further operations such as archiving.

See also

- “Work queues that use `StandardWorkItem`” on page 703

Define a typecode for a custom work queue

About this task

Before you begin developing the Gosu code for your custom work queue, you need to define the custom work queue type. For your custom type, you need to add a typecode for it in the `BatchProcessType` typelist. The constructor of your custom work queue class requires the typecode as an argument.

Procedure

1. Open Guidewire Studio
2. In the Studio **Project** window, expand **configuration > config > Extensions > Typelist**.
3. Open `BatchProcessType.ttx`.
4. Click the green plus button, , to add a typecode.
5. In the panel of fields on the right, specify appropriate values for the following fields:

code Specify a code value that uniquely identifies your work queue. This code value identifies the work queue in configuration files.

name Specify a human readable identifier for your work queue. This name appears for the work queue writer on the Server Tools **Batch Process Info** screen.

desc Provide a short description of what your custom process accomplishes. This description appears for your work queue on the Server Tools **Batch Process Info** screen.

Define a custom work item type

Before you begin

Before starting to define a custom work item type, review “Work queues and work item entity types” on page 702.

Procedure

1. Open Guidewire Studio.
2. Add a new entity for your custom work item type:
 - a) In the Studio **Project** window, expand **configuration > config > Extensions > Entity**.
 - b) Right-click **Entity**, and then select **New > Entity**.
 - c) In the **Entity** dialog box, enter the appropriate values.

For example, if creating a new work item entity called `MyWorkItem`, enter the following values:

Field	Example value
Entity	<code>MyWorkItem</code>
Entity Type entity	
Desc	Custom work item
Table	<code>myworkitem</code>
Type	keyable

Accept all the other default values in the dialog box.

- d) Click **OK**.
3. Set the entity to implement the correct work item delegate:
 - a) In the toolbar, in the drop-down list next to the green plus button, , select **implementsEntity**.

- b) In the panel of fields on the right, select **WorkItem** from the **name** drop-down list.
4. Add a soft reference to the unit of work for your custom work queue:
- Select the **column** field type in the drop-down list next to the green plus button, . Studio inserts a new **column** element in the element table.
 - In the panel of fields on the right, enter the entity name for the unit of work for the **name** value. For example, if the unit of work is an instance of an **Activity** object, add the following values at the panel at the right:

Field	Example value
name	Activity
type	Activity
nullOk	false

5. (Optional) Add other fields to your custom entity as meets your business needs.

What to do next

Whenever you define a custom work item for a work queue, your custom work queue class must implement the `createWorkItem` method to write the work item to the queue.

Creating a custom work queue class

After you define a typecode for your custom work queue and possibly create a custom work item type for it, you are ready to create your custom work queue class. This class contains the programming logic for the writer and the workers of your work queue. You must derive your class from `WorkQueueBase`, and you generally must override the following two methods.

`findTargets` Logic for the writer, which selects units of work for a batch and returns an iterator for the result set

`processWorkItem` Logic for the workers, which operates on a single unit of work selected by the writer

The `WorkQueueBase` provides other methods that you can override such as `shouldProcessItem`. However, you can generally develop a successful custom work queue by overriding the two required methods `findTargets` and `processWorkItem`.

WARNING: Do not implement multi-threaded programming in custom work queues derived from `WorkQueueBase`.

See also

- “Bulk insert work queues” on page 710

Custom work queue class declaration

In the declaration of your custom work queue class, you must include the `extends` clause to derive your work queue class from `WorkQueueBase`. Because the base class is a template class, your class declaration must specify the entity types for the unit of works and for the work items in your work queue.

The following example code declares a custom work queue type that has `Activitiy` as the target, or unit of work, type. It also declares that `StandardWorkItem` as the work queue type.

```
class MyWorkQueue extends WorkQueueBase <Activity, StandardWorkItem> { ... }
```

Custom work queue class constructor

You must implement a constructor in your custom work queue class. The constructor that you implement calls the constructor in the `WorkQueueBase` class. Your constructor associates the custom work queue class at runtime with its typecode in the `BatchProcessType` typelist, its work queue item type, and its target type.

The following example code is the constructor for a custom work queue. It associates the class at runtime with its batch process typecode `MyWorkQueue`. The code associates the work queue with the `StandardWorkItem` entity type. So, the work queue shares its work queue table with many other work queues. The code associates the work queue with the `Activity` entity type as its target unit of work type. So, the writer and the workers operate on `Activity` instances.

```
construct () {
    super (typekey.BatchProcessType.TC_MYWORKQUEUE, StandardWorkItem, Activity)
}
```

Do not include any code in the constructor of your custom work queue other than calling the super class constructor.

Developing the writer for your custom work queue

In your custom work queue class, override the `findTargets` method that your custom work queue class inherits from `WorkQueueBase` to provide your specific logic for a writer task.

IMPORTANT: Do not operate on any of the units of work in a batch within your writer logic. Otherwise, process history statistics for the batch will be incorrect.

In the `findTargets` method logic, return a query builder iterator with the results you want as targets for the work items in a batch. ClaimCenter uses the iterator to write the work items to the work queue. The `findTargets` method is a template method for which you specify the target entity type, the same type for which you make a query builder object.

The following example code is a writer for a work queue that operates on `Activity` instances. The query selects activities that have not been viewed for five days or more and returns the iterator.

```
override function findTargets (): Iterator <Activity> {
    // Query the target type and apply conditions: activities not viewed for 5 days or more
    var targetsQuery = Query.make(Activity)
    targetsQuery.compare(Activity#LastViewedDate, LessThanOrEquals, java.util.Date.Today.addBusinessDays(-5))
    return targetsQuery.select().iterator()
}
```

Returning an empty iterator

Generally, if your writer returns an iterator with items found by the query, ClaimCenter creates a `ProcessHistory` instance for the batch run. Sometimes the query in your writer finds no qualifying items. If your writer returns an empty iterator, ClaimCenter does not create a process history for that batch processing run.

Writing custom work item types

Suppose that you are creating a custom work queue class and decide not to use the `StandardWorkItem` work item and instead define a custom work item type. In you create your own class, you must override the `createWorkItem` method inherited from `WorkQueueBase` to write new work items to your custom work queue table. The `createWorkItem` method has two parameters.

- `target` – An object reference to an entity instance in the iterator returned from the `findTargets` method.
- `safeBundle` – A transaction bundle provided by ClaimCenter to manage updates to the work queue table.

Your method implementation must create a new work item of the custom work item type that you defined. Pass the `safeBundle` parameter in the new work item statement. Then, assign the `target` parameter to the target unit of work field that you defined for you custom work item type. If you defined additional fields, assign values to them, as well.

The return type for the `createWorkItem` method is any entity type that implements the `WorkItem` delegate. Return your new custom work item type.

The following Gosu example code creates a new custom work item that has a single custom field, an `Activity` instance. The implementation gives the target field in the parameter list the name `activity` to clarify the code. The custom work item type is `MyWorkItem`.

```
override function createWorkItem (activity : Activity, safeBundle : Bundle) : MyWorkItem {  
    var customWorkItem = new MyWorkItem(safeBundle)  
    customWorkItem.Activity = activity  
    return customWorkItem  
}
```

Developing workers for your custom work queue

In your custom work queue class, override the `processWorkItem` method that your custom work queue class inherits from `WorkQueueBase` to provide the programming logic for a worker task. The workers of a work queue are inherently single threaded. Do not attempt to improve the processing rate of individual workers by spawning threads from within your `processWorkItem` method.

The `processWorkItem` has a work item as its single parameter. You access the target, or unit of work, for the worker through the `WorkItem.Target` property. The type and target type of the work item are the types that you specified in the constructor of your custom work queue class derived from `WorkQueueBase`. At the same time that ClaimCenter calls the `processWorkItem` method, it sets the `Status` field of the work item to `checkedout`.

Successful work items

In your custom work queue class, return from the `processWorkItem` method after a worker finishes operating on a target instance. ClaimCenter then deletes the work item from the work queue.

The following example code extracts the targeted unit of work, an `Activity` instance, from the work item parameter. Then, the code sends the assigned user an email message.

```
override function processWorkItem (WorkItem : StandardWorkItem): void {  
  
    // Extract the unit of work: an Activity instance  
    var activity = extractTarget(WorkItem)  
  
    if (activity.AssignedUser.Contact.EmailAddress1 != null) {  
        // Send an email to the user assigned to the Activity.  
        mailUtil.sendEmailWithBody(null,  
            activity.AssignedUser.Contact, // To:  
            null, // From:  
            "Activity not viewed for five days", // Subject:  
            "Take a look at activity " + target.Subject + ", due on " + target.TargetDate + ".") // Body:  
    }  
  
    return  
}
```

In your override of the base `processWorkItem` method, specify the same work item entity type that you specified in the constructor of your custom work queue class.

Updates to entity data

The abstract `WorkQueueBase.processWorkItem` method does not have a bundle to manage database transactions related to targeted units of work. A bundle does exist at the time ClaimCenter calls your custom `processWorkItem` method, but ClaimCenter uses that bundle for updates to the work item. To modify entity data, you must create a bundle using the `Transaction.RunWithNewBundle` API in your custom work queue class.

The following example code uses the `RunWithNewBundle` transaction method to update the `EscalationDate` on the `Activity` instance from the work items that is processes.

```
uses gw.transaction.Transaction  
...  
override function processWorkItem (WorkItem : StandardWorkItem): void {
```

```

// Extract the unit of work: an Activity instance
var activity = extractTarget(WorkItem)

// Update the activity escalation date
Transaction.runWithNewBundle( \ bundle -> {
    activity = bundle.add(activity)           // add the activity to the new bundle
    activity.EscalationDate = java.util.Date.Today } ) // update the escalation date

return
}

```

See also

- *Gosu Reference Guide*

Managing failed work items

If your custom worker code encounters an error while operating on a target instance, throw an exception. ClaimCenter detects and throws some types of exceptions automatically, such as a concurrent data change exception (CDCE). These types of exceptions generally resolve themselves quickly and automatically. However, it is also possible that your code detects other logic errors that require human intervention to resolve. If so, implement a custom exception and throw it whenever your worker code detects the exceptional situation.

ClaimCenter catches exceptions thrown by code within the scope of your custom `processWorkItem` method. Whenever ClaimCenter catches an exception, it increments the `Attempts` property on the work item. If the value of the `Attempts` property does not exceed the value of the `WorkItemRetryLimit` parameter set in `config.xml`, ClaimCenter sets the `Status` property of the work item to `available`. Otherwise, ClaimCenter sets the `Status` property of the work item to `failed`.

Retrying work items that cause exceptions

ClaimCenter makes work items that trigger exceptions available again on the work queue, because many exceptions resolve themselves quickly. For example, a CDCE exception often occurs whenever two worker tasks attempt to update common data related to their two separate units of work. By the time ClaimCenter gives a work item that encountered an exception to another worker task, the exceptional condition often resolves itself. The second worker then process the work item to completion successfully.

Handling exceptions

In your custom work queue class, provide an implementation of the `handleException` method inherited from `WorkQueueBase` to augment the actions taken whenever code in the `processWorkItem` method throws an exception. For example, it is possible for your worker code to throw an exception whenever it encounters a logic error that cannot resolve itself without human intervention.

Bulk insert work queues

Guidewire recommends that you implement a custom work queue class that extends `BulkInsertWorkQueueBase`, rather than `WorkQueueBase` if it is possible for a query to determine the work items to create.

The use of the `BulkInsertWorkQueueBase` class provides the following performance optimizations:

- The work queue writer does not add duplicate entries to the work queue.
- The use of the bulk insert method eliminates the need to fetch and commit each work item individually.

As a consequence, a work queue that subclasses `BulkInsertWorkQueueBase` performs its work much more quickly than a work queues that subclasses `WorkQueueBase`.

Overview of bulk insert work queues

If your type of process works extremely large work batches, it is possible for the `WorkQueueBase.findTargets` method to return an iterator that exceeds the memory capacity of the work queue server. In some cases, the method returns an iterator for batches that typically number hundreds of thousands of units of work.

Thus, Guidewire recommends that you implement your custom work queue as a class that extends `BulkInsertWorkQueueBase` instead of `WorkQueueBase`. In your class, implement the `BulkInsertWorkQueueBase.buildBulkInsertSelect` method for your query logic, instead of the `WorkQueueBase.findTargets` method.

ClaimCenter calls the `buildBulkInsertSelect` method with a query object that has no restrictions on the target entity type that you specify in the class constructor. However, bulk insert work queues support standard work items only. Thus, do not attempt to use custom fields on work items to pass any data other than target entity instances to the workers of your bulk insert work queue.

Use the query builder APIs in your code to add restrictions, including restrictions based on joins, that select the targets for a batch. After your code returns from `buildBulkInsertSelect`, ClaimCenter submits a `SELECT` statement based on the query object directly to the database. The database then uses its native bulk insert capabilities to insert standard work items for the batch directly into the standard work queue table.

Bulk insert work queues are suitable if you can reduce the selection criteria for the targeted units of work to a single query builder query object. Because ClaimCenter creates and inserts work items at the database level, it is not necessary to implement the following methods:

- `createWorkItem`
- `shouldProcessItem`

The `BulkInsertWorkQueueBase` base class is a subclass of `WorkQueueBase`, so you must also implement method `processWorkItem` for the worker logic of a bulk insert work queue.

Bulk insert work queue declaration and constructor

In the declaration of your bulk insert work queue class, you must include the `extends` clause to derive your work queue class from abstract class `BulkInsertWorkQueueBase`. Your custom class must include a constructor that calls the constructor in the base class using the `super` syntax. Your constructor associates your bulk insert work queue class at runtime with the following items:

- The work queue typecode in the `BatchProcessType` typelist
- The work queue item type
- The user under which the database transaction runs

The following example code declares a bulk insert work queue type and implements a constructor. Unlike a work queue class that you derive from `WorkQueueBase`, the constructor does not specify the entity type of the `Target` fields on the work items for the work queue.

```
class MyBulkInsertWorkQueue extends BulkInsertWorkQueueBase <User, StandardWorkItem> {  
    construct () {  
        super(BatchProcessType.TC_MYBULKINSERTWORKQUEUE, StandardWorkItem, User)  
    }  
    ...  
}
```

Querying for targets of a bulk insert work queue

You provide the query logic for the writer thread of a bulk insert work queue by overriding the abstract `buildBulkInsertSelect` method that your custom work queue class inherits from `BulkInsertWorkQueueBase`.

The following Gosu example code selects the units of work for a batch run, which are users with a status of On Vacation.

```
override function buildBulkInsertSelect(query : InsertSelectBuilder, args: List<Object>) {  
    query.SourceQuery.compare(User#VacationStatus.PropertyInfo.Name, Relop.Equals,  
        VacationStatusType.TC_ONVACATION )  
}
```

The following Gosu example code selects the units of work for a batch run, which are activities that no one has viewed in five days or more. Notice that this implementation of the method uses the `extractSourceQuery` method to generate the query result.

```
override function buildBulkInsertSelect(builder : Object, args: List<Object>) {
    extractSourceQuery(builder).compare( Activity#LastViewedDate.PropertyInfo.Name, Relop.LessThanOrEquals,
        java.util.Date.Today.addBusinessDays(-5) )
}
```

Eliminating duplicate items in bulk insert work queue queries

In constructing the query for your custom work queue class, Guidewire recommends that you include the following method, which excludes duplicate `StandardWorkItem` work items from the query:

```
excludeDuplicatesOnStandardWorkItem(java.lang.Object opaqueBuilder, boolean allowFailedDuplicates)
```

The `excludeDuplicatesOnStandardWorkItem` method takes the following parameters:

<code>opaqueBuilder</code>	The query builder to use.
<code>allowFailedDuplicates</code> (Boolean)	Whether to recreate a failed work item.

The `excludeDuplicatesOnStandardWorkItem` helper method excludes `StandardWorkItem` work items only. If you query on another work item type, use custom code similar to the following to exclude the duplicate work items:

```
builder.mapColumn(XXWorkItem#Target.PropertyInfo as IEntityPropertyInfo,
    XX#Id.PropertyInfo as IEntityPropertyInfo)

var subQuery = Queries.createQuery(XXWorkItem)
if allowFailedDuplicates subQuery.compare(XXWorkItem#Status.PropertyInfo.Name, Relop.NotEquals,
    WorkItemStatusType.TC_FAILED)

builder.SourceQuery.subselect(XX#Id.PropertyInfo.Name, CompareNotIn, subQuery,
    XXWorkItem#Target.PropertyInfo.Name)
```

Processing work items in a custom bulk insert work queue

You provide the programming logic for the writer thread of a bulk insert work queue by implementing the abstract `processWorkItem` method that your custom work queue class inherits from `BulkInsertWorkQueueBase`. The `processWorkItem` method takes a single argument, which is the work item type to use as the unit of work.

The following Gosu example code processes the units of work for a batch run, which are activities that no one has viewed in five days or more.

```
override function processWorkItem(workItem: StandardWorkItem) {
    // Extract an object reference to the unit of work: an Activity instance
    var activity = extractTarget(workItem) // Convert the ID of the target to an object reference

    if (activity.AssignedByUser.Contact.EmailAddress1 != null) {
        // Send an email to the user assigned to the activity
        EmailUtil.sendEmailWithBody(null, activity.AssignedUser.Contact,
            null, // To:
            "Activity not viewed for five days", // From:
            "See activity" + activity.Subject + ", due on " + activity.TargetDate + ".") // Subject:
            // Body:
    }
}
```

Example work queue that bulk inserts its work items

The following Gosu code is an example of a custom work queue that uses bulk insert to write work items for a batch to its work queue. This custom work queue class derives from the base class `BulkInsertWorkQueueBase` instead of from `WorkQueueBase`. The code provides an implementation of the `buildBulkInsertSelect` method instead of `findTargets` for its writer logic. The code provides an implementation of the `processWorkItemMethod` for its worker logic.

Bulk insert work queues like this one use the `StandardWorkItem` entity type for its work queue table. To use a custom work queue, you must implement `createBatchProcess` method to tell ClaimCenter in which table to insert the work items created from the SELECT statement.

The unit of work for this work queue is an activity that has not been viewed for five days or more. The process sends an email to the user assigned to the activity, and then it sets the escalation date on the activity to the current date. The process updates entity data, so the `processWorkItemMethod` uses the `runWithNewBundle` API.

```

uses gw.api.database.Query
uses gw.api.database.Relop
uses gw.api.email.EmailUtil
uses gw.processes.BulkInsertWorkQueueBase

/**
 * An example of a process implemented as a work queue that sends email
 * to assignees of activities that have not been viewed for five days or more.
 */
class MyOptimizedActivityEmailWorkQueue extends BulkInsertWorkQueueBase <Activity, StandardWorkItem> {

    /**
     * Let the base class register this type of custom batch processing by
     * passing the batch processing type, the entity type the work queue item, and
     * the target type for the units of work.
     */

    construct() {
        super ( BatchProcessType.TC_NOTIFYUNVIEWEDACTIVITIES, StandardWorkItem, Activity )
    }

    /**
     * Select the units of work for a batch run: activities that have not been
     * viewed for five days or more.
     */
    override function buildBulkInsertSelect(builder : Object, args: List<Object>) {
        excludeDuplicatesOnStandardWorkItem(builder, true);           // if using StandardWorkItem
        extractSourceQuery(builder).compare(
            Activity#LastViewedDate PropertyInfo.Name, Relop.LessThanOrEquals,
            java.util.Date.Today.addBusinessDays(-5) )
    }

    /**
     * Process a unit of work: an activity not viewed for five days or more
     */
    override function processWorkItem(workItem: StandardWorkItem) {
        // Extract an object referent to the unit of work: an Activity instance
        var activity = extractTarget(workItem) // Convert the ID of the target to an object reference

        if (activity.AssignedByUser.Contact.EmailAddress1 != null) {
            // Send an email to the user assigned to the activity
            EmailUtil.sendEmailWithBody(null, activity.AssignedUser.Contact,                                // To:
                null,                                            // From:
                "Activity not viewed for five days",           // Subject:
                "See activity" + activity.Subject + ", due on " + activity.TargetDate + ".") // Body:
        }
    }
}

```

Examples of custom work queues

Example work queue that extends WorkQueueBase

The following Gosu code is an example of a simple custom work queue. It uses the `StandardWorkItem` entity type for its work queue table. Its unit of work is an activity that no one has viewed for five days or more. The process simply sends an email to the user assigned to the activity. The process does not update entity data, so the `processWorkItemMethod` does not use the `runWithNewBundle` API.

Note: In general, Guidewire recommends that your custom work queue class extend `BulkInserWorkQueueBase` rather than `WorkQueueBase`. The following example is for illustration only.

```

uses gw.api.database.Query
uses gw.api.database.Relop
uses gw.api.email.EmailUtil
uses gw.processes.WorkQueueBase
uses java.util.Iterator

/**
 * An example of batch processing implemented as a work queue that sends email

```

```

 * to assignees of activities that have not been viewed for five days or more.
 */
class MyActivityEmailWorkQueue extends WorkQueueBase <Activity, StandardWorkItem> {

    /**
     * Let the base class register this type of custom process by
     * passing the process type, the entity type the work queue item, and
     * the target type for the units of work.
    */

    construct() {
        super (BatchProcessType.TC_NOTIFYUNVIEWEDACTIVITIES, StandardWorkItem, Activity )
    }

    /**
     * Select the units of work for a batch run: activities that have not been
     * viewed for five days or more.
    */

    override function findTargets() :Iterator <Activity> {
        // Query the target type and apply conditions: activities not viewed for 5 days or more
        var targetsQuery = Query.make(Activity)
        targetsQuery.compare( Activity#LastViewedDate.PropertyInfo.Name, Relop.LessThanOrEquals,
            java.util.Date.Today.addBusinessDays(-5) )
        return targetsQuery.select().iterator()
    }

    /**
     * Process a unit of work: an activity not viewed for five days or more
     */
    override function processWorkItem(WorkItem: StandardWorkItem) {
        // Extract an object referent to the unit of work: an Activity instance
        var activity = extractTarget(WorkItem) // Convert the ID of the target to an object reference

        if (activity.AssignedByUser.Contact.EmailAddress1 != null) {
            // Send an email to the user assigned to the activity
            EmailUtil.sendEmailWithBody(null,
                activity.AssignedUser.Contact,                                         //To:
                null,                                                               //From:
                "Activity not viewed for five days",                                //Subject:
                "See activity" + activity.Subject + ", due on " + activity.TargetDate + ".") //Body:
        }
    }
}

```

Notice that this work queue implementation uses a `findTargets` method to return an iterator of target objects on which to base the work items.

See also

- “Example work queue that bulk inserts its work items” on page 712

Example work queue for updating entities

The following Gosu code is an example of a custom work queue that updates entity data as part of processing a unit of work. It uses the `StandardWorkItem` entity type for its work queue table. Its unit of work is an activity that has not been viewed for five days or more. The process sends an email to the user assigned to the activity, and then it sets the escalation date on the activity to the current date. The process updates entity data, so the `processWorkItemMethod` uses the `runWithNewBundle` API.

```

uses gw.api.database.Query
uses gw.api.database.Relop
uses gw.api.email.EmailUtil
uses gw.processes.WorkQueueBase
uses gw.transaction.Transaction
uses java.util.Iterator

/**
 * An example of batch processing implemented as a work queue that sends email
 * to assignees of activities that have not been viewed for five days or more.
 */
class MyActivityEscalationWorkQueue extends WorkQueueBase <Activity, StandardWorkItem> {

    /**
     * Let the base class register this type of custom process by
     * passing the process type, the entity type the work queue item, and
     * the target type for the units of work.
    */

    construct () {
        super (typekey.BatchProcessType.TC_NOTIFYUNVIEWEDACTIVITIES, StandardWorkItem, Activity)
    }
}

```

```

/**
 * Select the units of work for a batch run: activities that have not been
 * viewed for five days or more.
 */
override function findTargets (): Iterator <Activity> {
    // Query the target type and apply conditions: activities not viewed for 5 days or more
    var targetsQuery = Query.make(Activity)
    targetsQuery.compare(Activity#LastViewedDate, Relop.LessThanOrEquals,
        java.util.Date.Today.addBusinessDays(-5))
    return targetsQuery.select().iterator()
}

/**
 * Process a unit of work: an activity not viewed for five days or more
 */
override function processWorkItem (WorkItem : StandardWorkItem) {
    // Extract an object referent to the unit of work: an Activity instance
    var activity = extractTarget(WorkItem)

    // Send an email to the user assigned to the activity.
    if (activity.AssignedUser.Contact.EmailAddress1 != null) {
        EmailUtil.sendEmailWithBody(null,
            activity.AssignedUser.Contact, // To:
            null, // From:
            "Activity not viewed for five days", // Subject:
            "See activity " + activity.Subject + ", due on " + activity.TargetDate + ".") // Body:
    }

    // Update the escalation date on the assigned activity
    Transaction.runWithNewBundle( \ bundle -> {
        activity = bundle.add(activity) // Add the activity to the new bundle
        activity.EscalationDate = java.util.Date.Today } ) // Update the escalation date
}
}

```

Example work queue with a custom work item type

The following example creates a custom work queue that uses a custom work item type for its work queue table. The name of the custom work item type is `MyWorkItem`. The code provides an implementation of the `createWorkItem` method in addition to the `findTargets` method to add work items for a batch to the work queue.

The unit of work for the work queue is an activity that has not been viewed for five days or more. The process sends an email to the user assigned to the activity, and then it sets the escalation date on the activity to the current date.

See also

- “Custom work queues” on page 701

Define custom work item `MyWorkItem`

It is necessary to create a custom work item if using a custom work queue.

Before you begin

Before creating this custom work item, review “Example work queue with a custom work item type” on page 715. If necessary, also review “Custom work queues” on page 701.

Procedure

- Open Guidewire Studio.
- In the Studio **Project** window, expand **configuration > config > Extensions > Entity**.
- Right-click **Entity** and select **New > Entity**.
- Enter the following information in the **Entity** dialog and click **OK**.

Entity	MyWorkItem
Entity Type	Entity
Description	Custom work item

Type	keyable
------	---------

Note: See “Considerations for developing a custom work queue” on page 704 for a discussion as to why Guidewire recommends the use of `softentityreference` instead of `foreignkey` here.

For all other fields, accept that dialog defaults.

5. Using the add functionality at the top of the left-hand column (the plus sign), add the following subelements to the `MyWorkItem` entity.

Element	Element attribute
<code>column</code>	<ul style="list-style-type: none"> • <code>name</code> – <code>Activity</code> • <code>type</code> – <code>softentityreference</code> • <code>nullok</code> – <code>false</code>
<code>implementsEntity</code>	<ul style="list-style-type: none"> • <code>name</code> – <code>Workitem</code>

What to do next

To use this custom work item, implement custom work queue `MyActivitySetEscalationWorkQueue` as defined in “Implement custom work queue `MyActivitySetEscalationWorkQueue`” on page 716.

Implement custom work queue `MyActivitySetEscalationWorkQueue`

To create a custom work queue, implement a class that extends class `WorkQueueBase`.

Before you begin

Before creating this class, create custom work item `MyWorkItem` as defined in “Define custom work item `MyWorkItem`” on page 715.

Procedure

1. Open Guidewire Studio
2. In the Studio **Project** window, expand **configuration > gsrc**.
3. Create a new package directory named **workflow**.
4. Within the **workflow** folder, create a new Gosu class named `MyActivitySetEscalationWorkQueue`.
5. Populate this class with the following code.

```
package workflow

uses gw.processes.WorkQueueBase
uses gw.api.database.Query
uses gw.api.database.Relop
uses java.util.Iterator
uses gw.transaction.Transaction
uses gw.api.email.EmailUtil
uses gw.pl.persistence.core.Bundle

/**
 * An example of batch processing implemented as a work queue that sends email
 * to assignees of activities that have not been viewed for five days or more.
 */
class MyActivitySetEscalationWorkQueue extends WorkQueueBase <Activity, MyWorkItem> {

    /**
     * Let the base class register this type of custom process by
     * passing the process type, the entity type the work queue item, and
     * the target type for the units of work.
     */
    construct () {
        super (typekey.BatchProcessType.TC_NOTIFYUNVIEWEDACTIVITIES, MyWorkItem, Activity)
    }

    /**

```

```

    * Select the units of work for a batch run: activities that have not been
    * viewed for five days or more.
    */
override function findTargets (): Iterator <Activity> {
    // Query the target type and apply conditions: activities not viewed for 5 days or more
    var targetsQuery = Query.make(Activity)
    targetsQuery.compare(Activity#LastViewedDate.PropertyInfo.Name, Relop.LessThanOrEquals,
        java.util.Date.Today.addBusinessDays(-5))
    return targetsQuery.select().iterator()
}

/**
 * Write a custom work item
 */
override function createWorkItem (activity : Activity, safeBundle : Bundle) : MyWorkItem {
    var customWorkItem = new MyWorkItem(safeBundle)
    customWorkItem.Activity = activity
    return customWorkItem
}

/**
 * Process a unit of work: an activity not viewed for five days or more
 */
override function processWorkItem (workItem : MyWorkItem): void {
    // Get an object reference to the activity
    var activity = workItem.Activity

    // Send an email to the user assigned to the activity
    if (activity.AssignedUser.Contact.EmailAddress1 != null) {
        EmailUtil.sendEmailWithBody(null,
            activity.AssignedUser.Contact           // To:
            null,                                // From:
            "Activity not viewed for five days", // Subject:
            "See activity " + activity.Subject + ", due on " +
            activity.TargetDate + ".")           // Body:
    }

    // Update the escalation date on the assigned activity
    Transaction.runWithNewBundle( \ bundle -> {
        activity = bundle.add(activity)           // add the activity to the new bundle
        activity.EscalationDate = java.util.Date.Today } ) // update the escalation date
}

return
}
}

```

Example of the real-time creation of work items

Generally, ClaimCenter generates work items through the use of a writer in an implementation of the `createWorkItem` method on class `WorkQueueBase`. A batch job typically triggers the action of the writer.

However, there are situations in which it is better to create work items in real time, for example, while executing code in a Gosu rule. In any case, if the event point is already known, it does not make sense to perform additional queries about the event in a batch job.

The following example illustrate the steps involved in creating work items in real time that update the claim description in the ClaimCenter database. Whenever a user updates the description of a claim within ClaimCenter:

- ClaimCenter calls the Claim Preupdate rule set.
- Preupdate rule logic creates a work item for the update action.
- Periodically, at a time interval frequency defined by its `maxpolinterval` value, a worker scans the queue for new work items.
- The worker retrieves the new work item.
- The worker performs the work.

Note: It is important to understand that the bundle used during work item creation is part of the current transaction. Thus, if the code throws an exception in the Preupdate rule as part of the creation of the work item, ClaimCenter does not persist the work item to the database.

This example uses the following steps:

1. “Create a `ProcessWorkItem` entity” on page 718

2. “Add custom typecodes to the BatchProcessType typelist” on page 719
3. “Create a preupdate rule for processing work items” on page 719
4. “Create custom ProcessWQ class” on page 720
5. “Update the work queue configuration file” on page 721
6. “Test your custom work queue” on page 721

Create a ProcessWorkItem entity

This procedure creates a custom work item extension entity.

Before you begin

Before starting this procedure, read and understand the concepts outlined in “Example of the real-time creation of work items” on page 717.

Procedure

1. Open Guidewire Studio.
2. In the Studio **Project** window, expand **configuration > config > Extensions**.
3. Select **Entity** and right-click to open the context menu.
4. Select **New > Entity**.
5. In the **Entity** dialog, enter the following attribute information and press **OK**.

Entity	ProcessWorkItem
Entity Type entity	
Desc	Work item to use for processing Claim updates
Table	processworkitem
Type	keyable

6. Select **column** from the drop-down list near the top of the screen and click **+**.
7. Enter the following attribute information for the new column.

name	Claim
type	softentityreference
nullok	false

Note: See “Considerations for developing a custom work queue” on page 704 for a discussion as to why Guidewire recommends the use of **softentityreference** instead of **foreignkey** here.

8. Select the new entity row.
9. Using the right-click context menu, select the following:
Add new... > implementsEntity
10. Select the new **implementsEntity** row and then select **WorkItem** from the **name** drop-down list.
11. Save your work.

What to do next

Continue to “Add custom typecodes to the BatchProcessType typelist” on page 719.

Add custom typecodes to the BatchProcessType typelist

Before you begin

Before starting this procedure, perform the steps listed in “Create a ProcessWorkItem entity” on page 718.

About this task

This step adds a ProcessWQLoggers typecode to the BatchProcessType typelist.

Procedure

1. Open Guidewire Studio.
2. In the Studio **Project** window, expand **configuration > config > Extensions > Typelist**.
3. Open the **BatchProcessType** typelist for editing.
4. Select **typecode** from the drop-down list near the top of the screen and click **+**.
5. Enter the following attribute information for the new column.

```
code ProcessWQLoggers
name Process Work Item Loggers
desc Batch process type for processing work items
```

6. Select the newly created **typecode** row.
7. From the right-click context menu, select the following:
Add new... > typecode > category
8. Enter the following attribute information for the new category.

```
code UIRunnable
typelist BatchProcessTypeUsage
```
9. Select the typecode again and create a new category with the following attribute information.

```
code Schedulable
typelist BatchProcessTypeUsage
```
10. Save your work.

What to do next

Continue to “Create a preupdate rule for processing work items” on page 719.

Create a preupdate rule for processing work items

Before you begin

Before starting this procedure, perform the steps listed in “Add custom typecodes to the BatchProcessType typelist” on page 719.

About this task

This step creates a new Claim Preupdate rule that processes any work items associated with an update to a claim.

Procedure

1. Open Guidewire Studio.
2. In the Studio **Project** window, expand **configuration > config > Rule Sets > Preupdate > ClaimPreupdate**.
3. Select **ClaimPreupdate** in the center pane.
4. From the right-click context menu, select **New Rule** and enter the following information.

Rule name CPU9000 - Work Item Rules

5. Select the newly created CPU9000 rule, and create a new subrule with the following name.

Rule name CPU9010 - Process Work Item

6. Under **USES**, enter the following class name.

```
gw.api.system.CCLoggerCategory
```

7. Under **CONDITION**, enter the following return value.

```
return claim.isFieldChanged(Claim#Description)
```

8. Under **ACTION**, enter the following Gosu code.

```
CCLoggerCategory.RULE_HIERARCHY_ROOT.info("**** ENTERING " + actions.Rule.DisplayName + " Rule")
var workItem = new ProcessWorkItem(claim.getBundle())
workItem.Claim = claim.ID.Value
workItem.initialize()
```

9. Save your work.

What to do next

Continue to “Create custom ProcessWQ class” on page 720.

Create custom ProcessWQ class

Before you begin

Before starting this procedure, perform the steps listed in “Create a preupdate rule for processing work items” on page 719.

About this task

This step creates a Gosu class to process the claim update work item.

Procedure

1. Open Guidewire Studio.
2. In the Studio **Project** window, expand **configuration > gsrc**.
3. Select **gsrc** and select **New Package** from the right-click context menu.
4. For the new package name, enter **myWorkQueuePackage**, and click **OK**.
5. Select the new package, right-click and select **New > Gosu Class**.
6. Enter the following information.

Name myProcessWQClass

Kind Class

7. For class `myProcessWQClass`, enter the following code.

```
package myWorkQueuePackage

uses gw.processes.WorkQueueBase
uses gw.pl.persistence.core.Bundle
uses gw.api.system.CLoggerCategory

class myProcessWQClass extends WorkQueueBase <Claim, ProcessWorkItem> {

    construct() {
        super(BatchProcessType.TC_PROCESSWQLOGGERS, ProcessWorkItem, Claim)
    }

    override function processWorkItem(workItem: ProcessWorkItem) {
        CLoggerCategory.RULE_HIERARCHY_ROOT.info("*** PROCESSING workItem ${workItem.Claim} ***")
    }

    override function createWorkItem(claim: Claim, safeBundle: Bundle) : ProcessWorkItem {
        var workItem = new ProcessWorkItem(safeBundle)
        workItem.Claim = claim.ID.Value
        return workItem
    }
}
```

8. Save your work.

What to do next

Continue to “Update the work queue configuration file” on page 721.

Update the work queue configuration file

Before you begin

Before starting this procedure, perform the steps listed in “Create custom ProcessWQ class” on page 720.

About this task

This step adds a `ProcessWQLoggers` work queue to the work queue configuration file.

Procedure

1. Open Guidewire Studio.
2. In the Studio **Project** window, expand **configuration > config > workqueue**.
3. Open `work-queue.xml` for editing.
4. Add the following new work queue to the work queue configuration file.

```
<!-- My new work queue-->
<work-queue workQueueClass="myWorkQueuePackage.myProcessWQClass" progressinterval="60000">
    <worker instances="1" batchsize="250"/>
</work-queue>
```

5. Save your work.

What to do next

Continue to “Test your custom work queue” on page 721.

Test your custom work queue

Before you begin

Before starting this procedure, perform the steps listed in “Update the work queue configuration file” on page 721.

Procedure

1. Log into Guidewire ClaimCenter.

2. In the **Desktop**, select a claim.
3. Navigate to the claim **Loss Details** screen.
4. Click **Edit**.
5. Update the text in the **Description** field.
6. Click **Update** to save your work.
7. Verify that the following records exist in the rule execution log.

```
*** ENTERING CPU90010 - Process Work Item Rule
```

And, a few minutes later...

```
*** PROCESSING workItem 1 ***
```

Delayed processing of real-time work items

At times, it is useful to create work items in real time but to delay the processing of those work items until a later time. For example, suppose that you write a Gosu rule to create work items in response to a user-initiated event in ClaimCenter. However, you want to defer the processing of these work items until the middle of the night. At this point, the load on the processing server is lighter and you can use multiple worker threads to process the work items.

There are multiple technical reason for why you would want to do this, for example:

- The `processWorkItem` method that runs for each work item can require large amounts of processing power.
- The executor of each work queue performs a polling of the work queue multiple times during the day to determine if there are new work items. This work can place a heavy load on the work queue server.

Use scheduled batch process to execute work items

It is possible to delay processing a certain work item type by disabling the polling of the queue for those work items. A batch process, instead, notifies the queue workers that there is work available for processing at a later time.

Before you begin

Review the steps involved in creating work items in real time through Gosu rules before undertaking this example.

Procedure

1. Open Guidewire Studio.
2. Create a Gosu rule that creates work items in real time.
3. Create a class that extends `WorkQueueBase` to manage the work queue that processes these work items.
4. In this class, add a `createBatchProcess` method that creates a batch process to execute the work items created by the Gosu rule.

For example, you can do something similar to the following code in your work queue class.

```
package gwservices.myWorkQueuePackage
uses gw.api.webservice.cc.maintenanceTools.CCMaintenanceToolsImpl
uses gw.processes.BatchProcess
uses gw.processes.BatchProcessBase
uses gw.processes.WorkQueueBase
uses gw.pl.persistence.core.Bundle
uses gw.api.system.CCLoggerCategory

class myProcessWQClass extends WorkQueueBase <Claim, ProcessWorkItem> {

    construct() {
        super(BatchProcessType.TC_PROCESSWQLOGGERS, ProcessWorkItem, Claim)
    }

    override function processWorkItem(workItem: ProcessWorkItem) {
        CCLoggerCategory.RULE_HIERARCHY_ROOT.info("*** PROCESSING workItem ${workItem.Claim} ***")
    }
}
```

```

override function createWorkItem(claim: Claim, safeBundle: Bundle) : ProcessWorkItem {
    var workItem = new ProcessWorkItem(safeBundle)
    workItem.Claim = claim.ID.Value
    return workItem
}

/**
 * Associated batch process for this Workqueue.
 * No need to create WorkItems in the batch process as workitems are already created at run time in the Gosu rule
 * This batch file simply notifies the queue workers to process available workitems
 */
function createBatchProcess(args : Object[]): BatchProcess {
    return new BatchProcessBase(getQueueType()) {
        function doWork() {
            new CCMaintenanceToolsImpl().notifyQueueWorkers(getQueueType().getCode())
        }
    };
}

```

- Set the maximum polling interval for the work queue to 0 in file `work-queue.xml`.

The following example code sets the maximum polling interval to 0, which effectively disables work queue polling, and thus disables work item processing.

```

<!-- My new work queue -->
<work-queue workQueueClass="gw.myWorkQueuePackage.myProcessWQClass" progressinterval="60000">
    <worker instance="10" maxpollinterval="0" batchsize="1" />
</work-queue>

```

- Schedule the batch process created by your work queue class to run at a time of your choosing.

Set when work queue server is available for processing

It is possible to delay processing a certain work item type by setting when the work queue server that processes the work item is available. You do this by assigning the work queue to a specific server with a required work queue role, then bringing up that server only at specific times. In other words, you limit the work queue to certain servers that are active at specific times, for example, after the business day ends.

Before you begin

Review the steps involved in creating work items in real time through Gosu rules before undertaking this example.

Procedure

- Open Guidewire Studio.
- Open file `config.xml` for editing.
- Add a specific role for the work queue to the `<registry>` element and assign it to a specific server or servers. The following example code adds the `customRole` role to the `<registry>` element and assigns to server `prod2`.

```

<registry roles="..., customRole">
    <server serverid="prod2" roles="customRole"/>
</registry>

```

You can also assign the required `customRole` role to a server as you start the server from a command prompt.

- Define work queue parameters for the `customRole` server in file `work-queue.xml`.

```

<work-queue workQueueClass="gwservices.myWorkQueuePackage.myProcessWQClass" progressinterval="30000">
    <worker instances="3" server="#customRole"/>
</work-queue>

```

The hash mark in front of `customRole` indicates that the value that follows the hash mark is a server role and not a server ID.

- Start the server with the specified work queue role at a specific time, either manually or through some automated process.

Set when work items are available for processing

It is possible to delay processing a certain work item type by setting when that work item is available to perform work.

Before you begin

Review the steps involved in creating work items in real time through Gosu rules before undertaking this example.

Procedure

1. Open Guidewire Studio.
2. Create a Gosu rule that creates work items in real time.
For example, create a Gosu rule that creates work items to update activity events in real time.
3. Create a class that extends `WorkQueueBase` to process the work items created by the Gosu rule.
4. In this class, override the `createWorkItem` method and set the `availableSince` value to a future time.
For example, do something similar to the following code in your class.

```
override function createWorkItem(activity: Activity, safeBundle: Bundle) : ProcessWorkItem {
    var workItem = new ProcessWorkItem(activity.getBundle())
    workItem.Activity = activity.ID.Value
    workItem.initialize()

    //Set the 'available since' value to a future time
    var lng = DateUtil.currentDate().getToday().addHours(19).getTime()
    workItem.setAvailableSinceAsLong(lng)

    return workItem
}
```

This code delays processing of the work item until 1900 hours in a 24-hour clock.

Even though the affected work queue server does not process the work items until the specified time, the executor of the work queue continues to poll the queue throughout the day, even though no work is done.

Developing custom batch process

A batch process is code that runs without human intervention as background process on a single server instance to process the units of work in a batch sequentially.

IMPORTANT: Regardless of the volume of items or the duration of the processing window, Guidewire strongly recommends implementing any type of custom batch processing as a custom work queue.

See also

- “Examples of custom batch processes” on page 731
- “Categorizing a process typecode” on page 735

Custom batch process overview

Custom batch processes are Gosu classes that extend the `BatchProcessBase` base class. The `BatchProcessBase` base class includes code that ClaimCenter uses to manage the processing of your custom batch process class.

Note: Custom batch processes are not substitutes for shell scripts or batch files. Do not attempt to automate a batch of system commands by developing custom batch processes.

You extend the `BatchProcessBase` base class by providing your own implementation of the `doWork` abstract method that `BatchProcessBase` inherits from its super class, `SinglePhaseBatchProcess`.

Custom batch processes are inherently single threaded. Do not attempt to improve the throughput of units of work by spawning threads from within your `doWork` method. In particular, the inherited methods `incrementOperationsCompleted` and `incrementOperationsFailed` are not thread-safe. Entity instances in transactional bundles on separate threads can cause problems.

IMPORTANT: If your type of batch process requires processing units of work in parallel to achieve sufficient throughput, develop a custom work queue instead of a custom batch process.

WARNING: Do not implement multi-threaded programming in custom batch processes derived from `BatchProcessBase`.

Custom batch processes, bundles, and users

Most likely, any custom batch process that you create needs to create, delete, or modify one or more business objects. Your batch process class must create a new bundle to perform this work. If you do not explicitly associate a user with the bundle, ClaimCenter uses a default user for bundle creation and commit.

Do not use the Guidewire sys user (System User), or any other default user, for this purpose. Instead, create your own application system user and limit the permissions assigned to that user to what is appropriate to successfully perform the necessary operation. You then use the following syntax to create a bundle with a specific user.

```
gw.transaction.Transaction.runWithNewBundle(\ bundle -> BLOCK_BODY, user)
```

Create a custom batch process

About this task

Creating a custom batch process is a multistep process:

Procedure

1. In Studio, edit the `BatchProcessType` typelist and add a typecode to represent your custom batch process.
Note: Creating this new typecode adds support for the new batch process within the `BatchProcessInfo` PCF and the `IMaintenanceToolAPI` web service.
2. Create a class that extends the `BatchProcessBase` class (`gw.processes.BatchProcessBase`). The only method you must override is the `doWork` method, which takes no arguments. Perform your batch process work in this method.
3. In Studio, in the Plugins editor, register your new plugin for the `IProcessesPlugin` plugin interface.
4. To schedule your custom batch process to run regularly on a schedule instead of on-demand, add an entry for the batch process to the XML file `scheduler-config.xml`.

See also

- “Batch process base class” on page 726
- “Categorizing a process typecode” on page 735
- “Implementing `IProcessesPlugin`” on page 736
- *Administration Guide*

Define a typecode for a custom batch process

About this task

Before you begin developing the Gosu code for your custom batch process, you need to define the batch process type. For your custom type, you need to add a typecode for it in the `BatchProcessType` typelist. The constructor of your batch process implementation class requires the typecode as an argument.

Procedure

1. Open Guidewire Studio.

2. In the Studio **Project** window, expand **configuration > config > Extensions > Typelist**.
3. Open **BatchProcessType.ttx**.
4. Click the green plus button, , to add a typecode.
5. In the panel of fields on the right, specify appropriate values for the following fields:

code Specify a code value that uniquely identifies your custom batch process. This code value identifies the custom batch process in configuration files.

name Specify a human readable identifier for your custom batch process. This name appears for batch processing type on the Server Tools **Batch Process Info** screen.

desc Provide a short description of what your custom batch process accomplishes. This description appears for your batch process on the Server Tools **Batch Process Info** screen.

6. Select the newly added typecode.
7. Select **Add new... > typecode > category** from the right-click context menu.
This action adds a new **category** element under the selected typecode.
8. For this category, define the following fields:

typelist Select **BatchProcessTypeUsage** from the drop-down list.

code Select one of the available choices from the drop-down list. You must add at least one category or your batch process cannot run.

Results

Creating this new typecode adds support for the new batch process within the **BatchProcessInfo** PCF and the **IMaintenanceToolAPI** web service.

Batch process base class

Your custom batch process must extend the **BatchProcessBase** Gosu class. You must override the abstract **dowork** method, which takes no arguments. You can override other methods, but it is not strictly necessary because the base class defines meaningful defaults.

The **dowork** method does not have a bundle to track the database transaction. To modify data, you must use the **Transaction.RunWithNewBundle** method to create a bundle.

Ensure that your main **dowork** method frequently checks the **TerminateRequested** flag. If it is true, exit from your code. For example, if you are looping across a database query, exit from the loop.

IMPORTANT: ClaimCenter calls the **requestTermination** method in a different thread from the thread that runs the **dowork** method of your batch process.

Useful properties on class BatchProcessBase

The following list describes some of the useful properties on class **BatchProcessBase**.

Property	Description
DetailStatus	Returns the detailed status for the batch type. Class BatchProcessBase defines a simple default implementation. Override the default property getter to provide more useful detail information about the status of your batch process for the ClaimCenter Administration screens. The details might be important if your class experiences any error conditions. For Java implementations, you must implement this property getter as the method getDetailStatus , which takes no arguments.

Property	Description
Exclusive	<p>Sets whether another instance of this batch process can start while the current process is still running. The base class implementation of the <code>isExclusive</code> method always returns true. Override the <code>isExclusive</code> method if you need to customize this behavior. This value does not affect whether other batch process classes can run. It only affects the current batch process class.</p> <p>For maximum performance, be sure to set the property value to <code>false</code>, if possible. For example, if your batch process takes arguments in its constructor, it might be specific to one entity such as only a single <code>Claim</code> entity. If you want to permit multiple instances of your batch process to run in parallel, you must ensure your batch process class implementation returns <code>false</code>. For example,</p> <pre>override property get Exclusive() : boolean { return false }</pre> <p>For Java implementations, implement this property getter as the method <code>isExclusive</code>, which takes no arguments.</p>
Finished	<p>Returns a Boolean value to indicate whether the process completed. Completion says nothing about the errors, if any.</p> <p>For Java implementations, implement this property getter as the <code>isFinished</code> method.</p>
OperationsCompleted	<p>Returns a count of how many operations are complete, as an integer value.</p> <p>For Java implementations, implement this property getter as the <code>getOperationsCompleted</code> method.</p>
OperationsExpected	<p>Returns a count of how many operations ClaimCenter expects the batch process to perform, as an integer value.</p> <p>For Java implementations, implement this property as the <code>getOperationsExpected</code> method.</p>
OperationsFailed	<p>Returns an internal count for the number of operations that failed.</p> <p>For Java implementations, implement property getter as the <code>getOperationsFailed</code> method.</p>
Progress	<p>Dynamically returns the progress of the batch process. The base class returns text in the form <code>x of y</code>, with <code>x</code> being the amount of work completed and <code>y</code> being the total amount of work. If the value of <code>y</code> is unknown, the property merely returns <code>x</code>. ClaimCenter determines the values of <code>x</code> and <code>y</code> from the <code>OperationsExpected</code> and <code>OperationsCompleted</code> properties.</p> <p>For Java implementations, implement the property getter as the <code>getProgress</code> method.</p>
TerminateRequested	<p>Returns a Boolean value that is the return value from the <code>requestTermination</code> method.</p>
Type	<p>Returns the type of the current batch process. There is no need to override the default <code>BatchProcessBase</code> implementation of this property.</p> <p>For Java implementations, implement the property getter as the <code>getType</code> method, which takes no arguments.</p>

Useful methods on class BatchProcessBase

Class `BatchProcessBase` contains a number of useful methods.

checkInitialConditions

The `BatchProcessBase.checkInitialConditions` method has the following signature.

```
checkInitialConditions()
```

The `checkInitialConditions` method is a no-argument class method. It is possible for the batch process manager to call this method multiple times for the same batch job as it processes that job.

ClaimCenter instantiates batch process classes at startup on servers with the `batch` server role. Later, ClaimCenter calls the `checkInitialConditions` method to determine whether to start a batch process:

- If the method returns `true`, ClaimCenter starts the batch process by calling its `dowork` method.

- If the method returns `false`, the initial conditions are not met and ClaimCenter does not call the `doWork` method, skipping the batch process for the time being.

The base class implementation of the `checkInitialConditions` method always returns `true`. You must override this method if you want to provide a conditional response. If you override the `checkInitialConditions` method, be certain your code completes and returns quickly. Do not include long running code, such as queries of the database. The intent of the method is to determine environmental conditions, such as server run level. If you want to check initial conditions of data in the database, perform the query in the `doWork` method.

The batch process base class automatically ensures the server is at the maintenance run level if you configure your batch process typecode with the `MaintenanceOnly` category. You can perform additional checks of the current server run level by overriding the `checkInitialConditions` method.

If you override the `checkInitialConditions` method, forward the call to the superclass before returning. If the superclass returns `false`, you must return `false`. If the superclass returns `true`, then perform your additional checks of environmental conditions and return `true` or `false` appropriately.

Returning false from the `checkInitialConditions` method

Guidewire recommends that you do not return `false` from the `checkInitialConditions` method unless the condition that triggered the `false` return value is likely to be temporary. If your overridden `checkInitialConditions` method returns `false`, ClaimCenter:

- Freezes any on-going work in the batch process.
- Sets the **Last Run Status** value in the Server Tools Batch Process Info screen to **Running**.
- Executes the `checkInitialConditions` method once a minute unless the returned value becomes `true`, or you click **Stop** in the **Batch Process Info** screen for this batch process, or the repeat attempts time out.
- Stops executing the `checkInitialConditions` method after 10 minutes and generates the following error message in the ClaimCenter log.

`No one started batch process after 600 seconds.`

For example, suppose that your implementation of the `checkInitialConditions` method checks whether the necessary files are available for processing. In some cases, it can be necessary to schedule a batch process to run more often than the actual schedule of input files arriving for processing. In such a case, it is better for the `checkInitialConditions` method to return `true` to indicate that the batch job is ready to run. You then log an INFO or WARN message to the effect that the necessary files are missing.

getDescription

The `BatchProcessBase.getDescription` method has the following signature.

```
getDescription()
```

The `getDescription` method is a no-argument method that retrieves the description of the current batch type. The base class retrieves this string from the batch type typecode description. Override this method if you need to customize this behavior.

incrementOperationsCompleted

The `BatchProcessBase.incrementOperationsCompleted` method has the following signature.

```
incrementOperationsComplete()
```

The `incrementOperationsCompleted` method is a no-argument method that increments an internal counter for how many operations are complete. For example, suppose your batch process iterates across entities of some type with special conditions and specially handle each found item. For each entity you modify, call this method once to track the progress of this batch process.

This method returns the current number of operations that are complete. It is possible to use the returned value in the following ways:

- In the ClaimCenter user interface to show the progress of the batch process
- In debugging code to track the progress of the batch process

Note: For each entity for which you have an error condition, call this method once to track the progress of this batch process.

incrementOperationsFailed

The `BatchProcessBase.incrementOperationsFailed` method has the following signature.

```
incrementOperationsFailed()
```

The `incrementOperationsFailed` method is a no-argument method that increments an internal counter for how many operations failed. For example, suppose your batch process iterates across entities of some type with special conditions and specially handle each found item. For each entity for which you have an error condition, call this method once to track the progress of this batch process.

This method returns the current number of operations failed. It is possible to use the returned value in the following ways:

- In ClaimCenter to show the progress of the batch process
- In debugging code to track the progress of the batch process

IMPORTANT: For any operations that fail, call both the `incrementOperationsFailed` method and the `incrementOperationsCompleted` method. Also, call the `incrementOperationsFailedReasons` method at the same time as well.

incrementOperationsFailedReasons

The `BatchProcessBase.incrementOperationsFailedReasons` method has the following signature.

```
incrementOperationsFailedReasons(String msg)
```

The `incrementOperationsFailedReasons` method takes a single `String` argument that states the reason that a particular operation failed. Calling this method adds that reason to the `operationsFailedReasons` array property. Use public method `getOperationsFailedReasons` to retrieve the `operationsFailedReasons` array.

If you call the `incrementOperationsFailed` method, also call the `incrementOperationsFailedReasons` at the same time so that the operation of these two methods are in synchronization.

requestTermination

The `BatchProcessBase.requestTermination` method has the following signature.

```
requestTermination()
```

The `requestTermination` method is a no-argument class method.

Whenever you click the **Stop** button on the Server Tools **Batch Process Info** screen, ClaimCenter calls the `requestTermination` method on that batch process to terminate the batch process, if possible.

Your custom batch process must shut down any necessary systems and stop your batch process if you receive this message. If you cannot terminate your batch process, return `false` from this method.

The `BatchProcessBase` class implementation of the `requestTermination` method always returns `false`, which means that the request did not succeed. The base class also sets an internal `TerminateRequested` flag that you can check to see if a terminate request was received.

IMPORTANT: ClaimCenter calls the `requestTermination` method in a different thread from the thread that runs your batch process `doWork` method.

For typical implementations, use the following pattern:

- Override the `requestTermination` method and have it return `true`. Whenever a user requests termination of the batch process, ClaimCenter calls your overridden version of the method.
- Ensure that the `dowork` method in your custom class frequently checks the value of the `TerminateRequested` flag. In your `dowork` code, exit from your code if that flag is set. For example, if you are looping across a database query, exit from the loop and return.

Override the `requestTermination` method and return `false` if you genuinely cannot terminate the process soon. Be warned that if you do this, you risk the server shutting down before your process completes.

WARNING: Although you can return `false` from `requestTermination`, Guidewire strongly recommends you design your batch process so that you actually can terminate the action. It is critical to understand that returning either value does not prevent the application from shutting down or reducing run level. ClaimCenter delays shutdown or change in run level for a period of time. However, eventually the application shuts down or reduces run level independent of this batch process setting. For maximum reliability and data integrity, design your code to frequently check and respect the `TerminateRequested` property.

ClaimCenter writes a line to the application log to indicate whether it is possible to terminate the batch process. In other words, the log line includes the result of your `requestTermination` method.

If your batch process can only run one instance at a time, returning `true` does not remove the batch process from the internal table of running batch processes. This means that another instance cannot run until the previous one completes.

setChunkingById

The `BatchProcessBase.setChunkingById` method has the following signature.

```
setChunkingById(IQueryResult queryResult, int chunkSize)
```

The `setChunkingById` method returns nothing.

The default ClaimCenter behavior for query builder result sets is to retrieve all entries from the database into the application server. However, retrieving a large data set as a single block can cause problems. To mitigate this issue in working with batch processes, use the `setChunkingById` method to set the query retrieve chunk size.

After setting the chunking factor, the batch process iterates the Gosu query as usual. Beneath the surface, the query retrieves the data in chunks, using a series of separate SQL queries, rather than retrieving the data all at once in one massive block. The effect is very similar to the existing `setPageSize` method on the query API. However, the differences between the two methods include the following:

- Method `Query.select().setPageSize` uses `ROWNUM` (or its equivalent) to do the chunking.
- Method `BatchProcessBase.setChunkingById` orders the results by ID and uses `WHERE ID > lastChunkMaxId` to do the chunking.

Using ID chunking is much more robust than using the `setPageSize` method if your process alters the input to the original query, as batch processes often do. For example, suppose that your batch process looks for all unprocessed items of type X and then processes them. If you use `ROWNUM` chunking then the query for the first chunk functions correctly:

```
WHERE ROWNUM < chunksize
```

After processing the returned items, the Gosu query will issue another SQL query:

```
WHERE ROWNUM >= chunksize AND ROWNUM < chunksize * 2
```

Unfortunately, this query skips `chunksize` unprocessed items because processing the first chunk of data alters the input to the subsequent query.

In general, the use of ID chunking is not useful if the query `SELECT` statement selects for certain specific columns only. Row-based chunking is frequently more useful in querying entities, if there is no change to the entities that affects subsequent queries.

The following code sample illustrates the use of the `setChunkingById` method.

```
uses gw.processes.BatchProcessBase
uses gw.processes.ProcessHistoryPurge
uses gw.transaction.Transaction

class ChunkingTestBatch extends BatchProcessBase {
    construct() {
        super(BatchProcessType.TC_TESTCHUNKING)
    }

    private var _daysOld = 5
    private var _batchSize = 1024

    override final function doWork(): void {
        var query = new ProcessHistoryPurge().getQueryToRetrieveOldEntries(_daysOld)

        setChunkingById(query, _batchSize)
        OperationsExpected = query.getCount()

        var itr = query.iterator()

        while (itr.hasNext() and not TerminateRequested) {
            Transaction.runWithNewBundle(b -> {
                var cnt = 0
                while (itr.hasNext() and cnt < _batchSize and not TerminateRequested) {
                    cnt = cnt + 1
                    incrementOperationsCompleted()
                    b.delete(itr.next())
                }
            }, "su")
        }
    }
}
```

In this code, notice the use of the following `BatchProcessBase` properties and methods:

- Property `OperationsExpected`
- Property `TerminateRequested`
- Method `setChunkingById`
- Method `incrementOperationsCompleted`

Notice also that you must create a `TestChunking` typecode on `BatchProcessType`.

See also

- “Useful properties on class `BatchProcessBase`” on page 726
- “`incrementOperationsCompleted`” on page 728
- “`incrementOperationsFailed`” on page 729
- “Setting the page size for prefetching query results” on page 804

Examples of custom batch processes

This topic contains the following example batch processes:

- “Example batch process for a background task” on page 731
- “Example batch process for unit of work processing” on page 732

See also

- “Developing custom batch process” on page 724

Example batch process for a background task

The following Gosu code is an example of a custom batch process that operates as a background task instead of one that operates on a batch of units of work. Its process history does not track the number of items that processed successfully or that failed, because it has no formal units of work.

The following Gosu batch process purges old workflows. It takes one parameter that indicates the number of days for successful processes. Pass this parameter in the constructor to this batch process class. In other words, your implementation of `IProcessesPlugin` must pass this parameter such as the new `MyClass(myParameter)` when it instantiates the batch process. If the parameter is missing or `null`, it uses a default system setting.

Notice also that the second constructor uses the `TC_PURGEWORKFLOWS` typecode on the `BatchProcessType` typelist. If this typecode does not exist, you must create it.

```
package gw.processes

uses gw.processes.BatchProcessBase
uses java.lang.Integer
uses gw.api.system.PLConfigParameters
uses gw.api.admin.WorkflowUtil

class PurgeWorkflows extends BatchProcessBase {
    var _succDays = PLConfigParameters.WorkflowPurgeDaysOld.Value

    construct() {
        this(null)
    }

    construct(arguments : Object[]) {
        super(typekey.BatchProcessType.TC_PURGEWORKFLOWS)
        if (arguments != null) {
            _succDays = arguments[0] != null ? (arguments[0] as Integer) : _succDays
        }
    }

    override function doWork() : void {
        WorkflowUtil.deleteOldWorkflowsFromDatabase( _succDays )
    }
}
```

IMPORTANT: You must use the `runWithNewBundle` API if want to modify entity data in your custom batch process.

See also

- “Define a typecode for a custom work queue” on page 706
- *Gosu Reference Guide*

Example batch process for unit of work processing

The following Gosu code is an example of a type of a batch process that processes units of work rather than operating as a background task. Its process history tracks the number of items that were processed successfully or that failed. Its units of work are urgent activities.

The batch process implements a notification scheme for urgent activities such that:

- The activity owner must respond to an urgent activity within 30 minutes.
- If the activity owner does not handle the issue within that time limit, the batch process notifies a supervisor.
- If the supervisor fails to resolve the issue in 60 minutes, the batch process sends a message further up the supervisor chain.

If there are no qualified activities, the batch process returns `false` so that it does not create a process history. If there are items to handle, the batch process increments the count. The application uses this count to display batch process status as “`n of t`” or a progress bar. If there are no contact email addresses, the task fails and the application flags it as a failure.

This example checks the value of the `TerminateRequested` flag to terminate the loop if the user or the application requested to terminate the process.

Notice also, that if the a `TestBatch` typecode on the `BatchProcessType` typelist does not exist, you must create it.

The code in this Gosu example does not actually send the email. Instead, the code prints the email information to the console. You can change the code to use real email APIs.

```
package sample.processes

uses gw.api.database.Query
uses gw.api.email.Email
uses gw.api.email.EmailContact
uses gw.api.email.EmailUtil
uses gw.api.profiler.Profiler
uses gw.api.profiler.ProfilerTag
uses gw.api.util.DateUtil
uses gw.processes.BatchProcessBase
uses java.lang.StringBuilder
uses java.util.HashMap
uses java.util.Map

class TestBatch extends BatchProcessBase {
    static var tag = new gw.api.profiler.ProfilerTag("TestBatchTag1")

    //If BatchProcessType typcode TestBatch does not exist, you must create it.
    construct() {
        super( BatchProcessType.TC_TESTBATCH )
    }

    override function requestTermination() : boolean {
        super.requestTermination() // set the TerminationRequested flag
        return true // return true to signal that we will attempt to terminate in our doWork method
    }

    override function doWork() : void { // no bundle
        var frame = Profiler.push(tag)

        try {
            var activityQuery = Query.make(Activity)
            activityQuery.compare(Activity#Priority, Equals, Priority.TC_URGENT)
            activityQuery.compare(Activity#Status, Equals, Activitystatus.TC_OPEN)
            activityQuery.compare(Activity#createTime, LessThan, DateUtil.currentDate().addMinutes(-30))
            OperationsExpected = activityQuery.select().getCount()
            var map = new HashMap<Contact, StringBuilder>()

            for (activity in activityQuery.select()) {
                if (TerminateRequested) {
                    return
                }
                incrementOperationsCompleted()
                var haveContact = false
                var msgFragment = constructFragment(activity)
                haveContact = addFragmentToUser(map, activity.AssignedUser, msgFragment) or haveContact
                var group = activity.AssignedGroup

                if (activity.CreateTime < DateUtil.currentDate().addMinutes( -60 )) {
                    while (group != null) {
                        group = group.Parent
                        haveContact = addFragmentToUser(map, group.Supervisor, msgFragment) or haveContact
                    }
                }

                if (!haveContact) {
                    incrementOperationsFailed()
                    addFragmentToUser(map, User.util.UnrestrictedUser, msgFragment)
                }
            }

            if (not TerminateRequested) {
                for (addressee in map.Keys) {
                    sendMail(addressee, "Urgent activities still open", map.get(addressee).toString())
                }
            } finally {
                Profiler.pop(frame)
            }
        }

        private function constructFragment(activity : Activity) : String {
            return formatAsURL(activity) + "\n\t" + "Subject: " + activity.Subject + " AssignedTo: "
                + activity.AssignedUser + " Group: " + activity.AssignedGroup + " Supervisor: "
                + activity.AssignedGroup.Supervisor + "\n\t" + activity.Description
        }

        private function formatAsURL(activity : Activity) : String {
            // TODO: YOU MUST ADD A PCF ENTRYPPOINT THAT CORRESPONDS TO THIS URL TO DISPLAY THE ACTIVITY.
            return "http://ClaimCenter:8080/cc/Activity.go(${activity.ID})"
        }

        private function addFragmentToUser(map : HashMap<Contact, StringBuilder>, user : User,
            msgFragment : String) : boolean {
            if (user != null) {

```

```

var email = user.Contact.EmailAddress1
if (email != null and email.trim().length > 0) {
    var sb = map.get(email)

    if (sb == null) {
        sb = new StringBuilder()
        map.put(user.Contact, sb)
    }

    sb.append(msgFragment)
    return true
}
return false
}

private function sendMail(contact : Contact, subject : String, body : String) {
    var email = new Email()
    email.Subject = subject
    email.Body = "<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">\n" +
        "<html>\n" +
        "  <head>\n" +
        "    <meta http-equiv=\"content-type\"\\n\" + \"content=\"text/html; charset=UTF-8\"\\n\" +\n" +
        "    <title>${subject}</title>\n" +
        "  </head>\n" +
        "  <body>\n" +
        "    <table>\n" +
        "      <tr><th>Subject</th><th>User</th><th>Group</td><th>Supervisor</th></tr>" +
        "    </table>" +
        "  </body>" +
        "</html>"
    email.addToRecipient(new EmailContact(contact))
    EmailUtil.sendEmailWithBody(null, email);
}
}

```

To use this entry point, create the following PCF entry point file with name **Activity**.

```

<PCF>
    <EntryPoint authenticationRequired="true" id="Activity" location="ActivityForward(actvtIdNum)">
        <EntryPointParameter locationParam="actvtIdNum" type="int"/>
    </EntryPoint>
</PCF>

```

In Studio, create this **Activity.pcf** file in the following location: **configuration > config > Page Configuration > pcf > entrypoints**

Also include the following **ActivityForward** PCF file as the **ActivityForward.pcf** in each place in the PCF hierarchy that you need it:

```

<PCF>
    <Forward id="ActivityForward">
        <LocationEntryPoint signature="ActivityForward(actvtIdNum : int)" />
        <Variable name="actvtIdNum" type="int"/>
        <Variable name="actvt" type="Activity">initialValue="find(a in Activity
            where a.ID == new Key(Activity, actvtIdNum))"/>
        <ForwardCondition action="ClaimForward.go(actvt.Claims);
            ActivityDetailWorksheet.goInWorkspace(actvt)"/>
    </Forward>
</PCF>

```

IMPORTANT: You must use the `runWithNewBundle` transaction method to modify entity data in your custom batch process. For more information, see the *Gosu Reference Guide*.

Working with custom processing

Before your custom processing can run, you must perform the steps outlined in the following topics:

- “Categorizing a process typecode” on page 735
- “Updating the work queue configuration” on page 735
- “Implementing IProcessesPlugin ” on page 736

The steps to perform depend on whether you implemented a custom work queue or a custom batch process.

See also

- *Administration Guide*

Categorizing a process typecode

For your type of custom process to run, you need to associate its typecode with appropriate categories in the `BatchProcessType` typelist. You must categorize the typecode whether you implemented a custom work queue or a custom batch process. You must add at least one runnable category or your type of custom batch processing cannot run.

The following list describes the valid runnable categories.

<code>UIRunnable</code>	The process is runnable both from the user interface and from the web service APIs.
<code>APIRunnable</code>	The process is runnable only from web service APIs.
<code>Schedulable</code>	It is possible to schedule the process to execute at specific dates and times.
<code>MaintenanceOnly</code>	Is only possible to run the process if the system is at maintenance run level.

The **Work Queue Info** screen shows your custom work queue, regardless how you categorize your process type code. Categorizing a process type code affects only whether the **Batch Process Info** screen displays custom batch processes and writers for work queues and their runnable characteristics.

See also

- “Define a typecode for a custom work queue” on page 706
- “Define a typecode for a custom batch process” on page 725
- To learn how to add categories to a typecode, see the *Configuration Guide*.

Updating the work queue configuration

ClaimCenter instantiates custom work queues that derive from `WorkQueueBase` or `BulkInsertWorkQueueBase` at start up, for servers in a ClaimCenter cluster with the `workqueue` server role. For ClaimCenter to instantiate your custom work queue, you must add a `work-queue` element for it in the `workqueue-config.xml` file.

The `work-queue` element in `workqueue-config.xml` has the following syntax.

```
<work-queue workQueueClass="string" progressinterval="decimal">
  <worker instances="1" />
</work-queue>
```

The `work-queue` element

The `work-queue` element identifies your custom work queue to ClaimCenter. For the `workQueueClass` attribute, specify the fully qualified class name of your custom work queue Gosu class.

The value of the `progressinterval` attribute is measured in milliseconds, and there is no default. It specifies the interval of time that ClaimCenter waits for a worker task to complete an allotment, or batch, of work items checked out to it from the work queue. If a worker does not complete an allotment within the time specified, the remaining work items become orphaned, and ClaimCenter assigns them to another worker task to complete.

An interval that is too short can result in many orphaned work items, which can create processing overhead by frequently reassigning work items. The default allotment, or batch size, is 10 work items. Start with a value of 600000 for `progressinterval`. If you see many orphaned work items, increase the value. A value that exceeds the actual interval required causes no processing delays.

The `worker` element

The XML schema requires a `worker` element for your custom work queue to operate. The `instances` attribute sets the number of work tasks to create initially for the work queue at server startup. Generally a value of 1 is sufficient.

See also

- *Administration Guide*

Implementing IProcessesPlugin

ClaimCenter instantiates custom batch processes that derive from `BatchProcessBase` at startup, for servers in a ClaimCenter cluster with the `batch` server role. Although ClaimCenter instantiates your class on all servers that have the `batch` server role, a single run of your custom batch process executes on only one of the servers.

For ClaimCenter to instantiate your custom batch process, your implementation of the `IProcessesPlugin` plugin must reference the typecode from the `BatchProcessType` typelist for your custom process. In the base configuration, ClaimCenter provides a `Gosu ProcessesPlugin` class that implements the `IProcessesPlugin` interface in the following location in Guidewire Studio™ for ClaimCenter:

`configuration > config > gsrc > gw > plugin > processes`

For each typecode in the `BatchProcessType` that represents a custom batch process, ClaimCenter calls the `createBatchProcess` method of the `Processes` plugin. The `createBatchProcess` method takes a typecode from the `BatchProcessType` as a parameter. The method uses a `switch` statement and a `case` clause to determine which process type to instantiate.

The following example code demonstrates how to implement the `IProcessesPlugin` plugin. The code instantiates the individual custom batch process in the `switch case` statements.

```
uses gw.plugin.processing.IProcessesPlugin
uses gw.processes.BatchProcess

@Export
class ProcessesPlugin implements IProcessesPlugin {
    construct() { }

    override function createBatchProcess(type : BatchProcessType, arguments : Object[]) : BatchProcess {
        switch(type) {
            case BatchProcessType.TC_PURGEMESSAGEHISTORY:
                return new PurgeMessageHistory(arguments)
            case BatchProcessType.TC_CUSTOMBATCHPROCESS:
                return new CustomBatchProcess()
            default:
                return null
        }
    }
}
```

Passing initialization parameters to batch process constructors

Every time that ClaimCenter executes a batch process, whether it is a scheduled process, or requested manually or through web services, ClaimCenter creates a new instance of the batch process class. Thus, it is important the class constructor not perform programming logic.

However, it is possible to pass one or more initialization parameters to a constructor that creates the batch process. For example, the implementation class for the `PurgeMessageHistory` batch process type contains two constructors, one that takes no arguments and one that does take arguments.

```
class PurgeMessageHistory extends BatchProcessBase {

    var _ageInDays : int

    construct() {
        this({CCConfigParameters.KeepCompletedMessagesForDays.Value})
    }

    construct(arguments : Object[]) {
        super(BatchProcessType.TC_PURGEMESSAGEHISTORY)
        if (arguments.length == 1 and arguments[0] typeis Integer) {
            _ageInDays = Coercions.toIntFrom(arguments[0])
        } else {
            _ageInDays = CCConfigParameters.KeepCompletedMessagesForDays.Value
        }
    ...
}
```

Notice, however, that the constructor with arguments performs no actual programming logic. It simply determines the value to use in initializing the batch process.

Starting processes with initialization parameters from command prompt

Use the `-startprocess` command, available with the `maintenance_tools` command prompt utility, to start a batch process. Using this command, it is possible to start the batch process with, or without, initialization parameters. Use the following syntax.

```
maintenance_tools -password password -startprocess process [-args arg1 arg2 ...]
```

To use arguments in instantiating custom batch processing, use the `createBatchProcess(type, args)` method on the `ProcessesPlugin` class. In this way, you can pass initialization parameters to the batch process constructor called by the new operation.

Starting processes with initialization parameters from web services

Class `MaintenanceToolsAPI` defines a `startBatchProcess` method that accepts initialization parameter and that returns a `ProcessID` value:

```
function startBatchProcessWithArguments(processName : String, arguments : String[]) : ProcessID {  
    return getDelegate().startBatchProcess( processName, arguments )  
}
```

Use this method to start a batch process with initialization parameters, for example:

```
processID = maintenanceToolsAPI.startBatchProcessWithArguments("myCustomBatchProcess", args)
```

The typecode for the batch processing type must include the `APIRunnable` category in order to start your custom batch process with the `MaintenanceToolsAPI` web service.

See also

- “Categorizing a process typecode” on page 735
- “Implementing `IProcessesPlugin`” on page 736
- “Using web service methods with batch processes” on page 124
- *Administration Guide*

Retrying failed batch process items

In the base configuration, ClaimCenter provides abstract class `RetryBatchProcess` that you can extend to create a custom batch process that retries items that failed during execution of the batch process.

Abstract class `RetryBatchProcess`

Class `RetryBatchProcess` provides an abstract extension class for `BatchProcessBase`, with the added ability that the batch process attempts to retry items that failed during the initial execution of the batch process. After the batch process completes its initial execution, the process retries all items that failed during the execution of the process. If any of the retried items still fail, the process does not retry those items again. Instead, the process logs the retry failure in the application log.

The default Guidewire implementation of the `RetryBatchProcess` class retries a failed item only if the failure was due to a concurrent data change exception. Class logic considers all other exceptions as unrecoverable failures.

Public methods on class `RetryBatchProcess`

Class `RetryBatchProcess` contains the following public methods. Some of these methods are themselves abstract.

getChunkSize()

The `getChunkSize` method returns the value set for chunking the results of the query used for fetching items for processing. The default value is 100. Guidewire recommends that you do not change this value without thought. Especially, do not change this value to either 0 or an extremely high number.

fetchItemsForProcessing()

The `fetchItemsForProcessing` method returns the objects for processing as an `IQueryBeanResult<T>` object. You must provide an overridden method body for this `abstract` class method in your custom batch process.

processItem(item)

The `processItem` method defines the actual work to perform on each individual batch item. The item for processing is the passed-in `item` value. You must provide an overridden method body for this `abstract` class method in your custom batch process.

Guidewire guarantees that class logic wraps each item for processing in a writable bundle and active transaction. Therefore, any implementation of this method need not try to create its own transaction or move the item to any other bundle.

doWork()

Guidewire marks the `doWork` class method as `protected` and `final`. It is not possible to override the Guidewire implementation of this class method.

The `doWork` method does the following:

1. It calls class method `fetchItemsForProcessing`, which returns a set of `IQueryBeanResult` objects.
2. It processes each returned item in its own bundle by calling private class method `processItemAndCommit` (which calls class method `processItem` on each item).
3. If the `processItem` method throws an exception, the `doWork` method calls class method `shouldRetryItem` on that item.
4. If the `shouldRetryItem` returns `true` for an item (which means the exception was a concurrent data change exception), the `doWork` method adds the item to the internal list of items for retry.
5. If the `shouldRetryItem` returns `false` (for all other exception types), the `doWork` method calls private method `handleUnrecoverableException` on the exception.
6. After the `doWork` method processes all fetched items, it calls class method `retryProcessing` on each failed item (which method calls the `processItem` method on each item again, by default).

shouldRetryItem(processingEx, item)

The `shouldRetryItem` method determines if it is possible to recover from an exception that happened during the processing of a specific item. The passed-in method arguments have the following meaning:

- `item` - The item that failed its initial processing.
- `processingEx` - The exception thrown during the initial processing of the item.

The method returns `true` if it is possible to recover from the exception, and thus, it is possible to retry processing the item. The method returns `false` otherwise.

In the default implementation, the method returns `true` only if the processing failure throws a concurrent data change exception. The method considers all other types of exceptions to be unrecoverable failures. It is possible for your custom implementation of this method to override this basic implementation logic.

prepareFailedItemForRetry(item)

The `prepareFailedItemForRetry` method determine if an item queued for retry is still eligible for processing. The `doWork` method calls the `prepareFailedItemForRetry` method during execution of that method.

It is necessary to call this method as it is possible for the item to have been modified externally, with those modifications rendering the item ineligible for reprocessing. The method returns one of the following:

- The object on which to retry processing
- `null`, if the method determination that it is not possible to retry processing on the object

In the default implementation, the method simply refetches the passed-in `item` from the application cache or database. Guidewire recommends that your custom implementation of this method override this basic implementation logic to impose specific eligibility requirements on the item. In your implementation of this class, call `super.prepareFailedItemForRetry`.

retryProcessing(item)

The `retryProcessing` method retries processing a specific item that had originally failed processing due to a recoverable exception. In the default implementation, the method retries processing the passed-in `item` only once. If the item processing fails again during the retry, the method considers the failure to be a permanent failure and does not attempt any further processing of the item. The method logs the retry failure to the application log.

It is possible for your custom implementation of this method to override this basic implementation logic. Be aware that the default implementation of this method calls an internal method to wrap the action in a bundle. The method then increments the number of completed operations. In your implementation of this class, call `super.retryProcessing`, if you can. Otherwise, you need to create your own bundle and track the number of completed operations in your code.

getItemDescription(item)

The `getItemDescription` method returns a human-friendly description (`String`) of the passed-in `item` for use in log statements.

Creating a custom retry batch process class

Your custom class must do the following:

- Extend abstract class `RetryingBatchProcess`.
- Provide overridden method bodies for all `RetryingBatchProcess` class methods declared `abstract`.
- Provide overridden method bodies for those default method implementations that do not meet your business needs.

Your custom Gosu class must include a class constructor of the following form:

```
construct () {
    //If BatchProcessType typcode CustomWorkQueue does not exist, you must create it.
    super (typekey.BatchProcesstype.TC_CUSTOMWORKQUEUE)
}
```

You must create a typecode in typelist `BatchProcessType` for your custom batch process.

Monitoring ClaimCenter processes

Although ClaimCenter processes run without human intervention as background tasks, it is important to manage the processes and monitor their progress. ClaimCenter provides a number of features for monitoring default and custom processes.

Monitoring ClaimCenter processing using administrative screens

Guidewire provides two Server Tools administration screens that you can use to manage and monitor batch processes and work queues. They are:

- **Batch Process Info** screen
- **Work Queue Info** screen

The Batch Process Info screen

System administrators can use the (Server Tools) **Batch Process Info** screen to start and view information about ClaimCenter processes, including writer processes for work queues.

The Work Queue Info screen

System administrators can use the (Server Tools) **Work Queue Info** screen to control and view information about ClaimCenter work queues. From this **Work Queue Info** screen, you can also download process history information on individual work queues.

See also

- *Administration Guide*

Monitoring ClaimCenter processes for completion

Night-time processing frequently requires chaining, meaning that the completion of one type of process starts another type of follow-on process. Regardless of implementation mode – work queue or batch process – you most likely must develop custom process completion logic for your type of nightly batch processing.

Characteristics of a completed process run vary depending on the implementation mode:

Work queues

Complete if no work items for a batch remain in the work queue, other than work items that failed

Batch processes

Complete if the batch process stopped and its process history is available

In the base ClaimCenter configuration, Guidewire provides an `IBatchCompletedNotification` interface that you can implement and call from the Process Completion Monitor process.

```
public interface IBatchCompletedNotification extends InternalPlugin {
    void completed(ProcessHistory var1, int var2);
}
```

The completed method takes the following parameters.

var1 Process history

var2 Number of failed work items

In your implementation of this interface, override the completed method to perform specific actions after a work queue or batch process completes a batch of work. Add a case clause to the method for your type of custom batch processing by specifying its typecode from the `BatchProcessType` typelist, for example:

```
switch(type) {
    case BatchProcessType.TC_ACTIVITYESC:
        return new ActivityEscalationWorkQueue()
    case BatchProcessType.TC_MYCUSTOMBATCHPROCESS:
        return new MyCustomBatchProcess()
    ...
    default:
        return null
}
```

See also

- *Administration Guide*

Monitoring batch processes by using maintenance tools

You can start, terminate, or retrieve the status of batch processes, including writers for work queues, by using the `maintenance_tools` command or the `MaintenanceToolsAPI` web service. The web service provides additional functions not available with the command, such as accessing and modifying configuration properties for work queues and notifying workers of work on the queue.

See also

- “Maintenance tools web service” on page 124

- *Administration Guide*

Periodically purging process entities

Entities related to processing, such as process history and work items, accumulate as each process run completes. Guidewire recommends that you periodically purge outdated entities for completed process work to avoid slowing the performance of all types of processing.

For details of the processes that you can use to purge outdated data, see the following:

- *Administration Guide*

Managing user entity updates in batch processing

It is possible for an update to the `User` entity to null out additional fields on the entity if the executing user is a ClaimCenter System user. This situation can arise during batch processing or during the execution of code in the Studio Gosu tester. For example, if the code sets the value of `User.ExternalUser` to `true`, ClaimCenter also changes a number of other fields within the entity automatically. These additional changes include:

- Setting the value of `User.UserType` to `Other`
- Setting the value of `User.Organization` to `null`
- Setting the `Address` fields on the associated `Contact` entity to `null`

Specify a Current User that belongs to the same organization as the User executing the code so that the User inherits the organization information of the Current User. The Current User must be an external user as well.

Set the external user field in a batch process

About this task

If you need to set the `User.ExternalUser` field in a batch process, Guidewire recommends that you take the following steps to manage this process.

Procedure

1. Set the `User.ExternalUser` property to the specified value.
2. Retrieve the value of Current User that the bundle specifies.
3. If the User is not an external user, use the existing organization to which the user belongs. In other words, do not clear this field.
4. If the User is an external user:
 - Remove all the carrier's internal roles for the User, meaning all roles for which `Role.isCarrierInternalRole` is `true`.
 - Set the User's organization to the Current User's organization.
5. If the User's organization is not the same as the Current User's organization, remove all the groups and producer codes associated with the User.
6. If the User's organization is not a carrier, set the `User.UseProducerCodeSecurity` property to `true`.
7. If the value of `User.UseOrgAddress` is `true`, set the user's contact address information to the organization address.
8. Set the `User.UserType` property to `Other`.

Querying and connecting to databases

The query builder APIs provide support for retrieving information from ClaimCenter application databases.

The API framework models features of SQL SELECT statements to make object-oriented Gosu queries. The query builder APIs support many, but not all, features of the SQL SELECT statement.

Connections to the database connection pool can be automatic or, to improve performance, reserved.

chapter 38

Query builder APIs

The query builder APIs provide support for retrieving information from ClaimCenter application databases.

The API framework models features of SQL SELECT statements to make object-oriented Gosu queries. The query builder APIs support many, but not all, features of the SQL SELECT statement.

Supported SQL features in query builder APIs

Query builder APIs provide functionality similar to these features of SQL SELECT statements.

Keyword or clause in SQL	Equivalent in query builder APIs example	Purpose
SELECT *	In Gosu: <pre>var query = Query.make(entity-type)</pre> In Java: <pre>Query<entity-type> query = Queries. createQuery(entity-type.TYPE)</pre>	Begins a query. Generally, the results of a query are a set of object references to selected entity instances.
FROM <i>table</i>	<pre>var query = Query.make(entity-type)</pre>	Declares the primary source of data.
DISTINCT	<pre>query.withDistinct(true)</pre>	Eliminates duplicate items from the results.
<i>column1 AS A[, column2 AS B [...]]</i>	<pre>query.select({ QuerySelectColumns.pathWithAlias("A", Paths.make(entity#field1)) [, QuerySelectColumns.pathWithAlias("B", Paths.make(entity#field2)) ...] })</pre>	Produces a result that contains a set of name/value pairs, instead of a set of entity instances.
JOIN <i>table</i> ON <i>column</i>	<pre>var table = query.join(entity#foreign-key)</pre>	Joins a dependent source of information to the primary source, based on a related column or field.
WHERE	<pre>query.compare(entity#field-name, parameters) query.compareIgnoreCase(entity#field-name, parameters)</pre>	Fetches information that meets specified criteria.

Keyword or clause in SQL	Equivalent in query builder APIs example	Purpose
	<pre>query.between(entity#field-name, parameters) query.compareIn(entity#field-name, parameters) query.compareNotIn(entity#field-name, parameters) query.startsWith(entity#field-name, parameters) query.contains(entity#field-name, parameters)</pre>	
ORDER BY <i>column1</i> [, <i>column2</i> [...]	<pre>var orderedResult = result.orderBy(QuerySelectColumns.path(Paths.make(entity#field1))) [.thenBy(QuerySelectColumns.path(Paths.make(entity#field2))) [...]]</pre>	Sort results by specific columns or fields.
GROUP BY	Implied by aggregate functions on fields	Return results grouped by common values in a field.
HAVING	query.having()	Return results based on values from aggregate functions.
UNION	var union = query1.union(query2)	<p>Combine items fetched by two separate queries into a single result.</p> <p>Note: You cannot use the <code>union</code> method on query objects passed as arguments to the <code>subselect</code> method.</p>
FIRST	result.FirstResult	Limit the results to the first row, after grouping and ordering.
TOP LIMIT	<pre>result.getCountLimitedBy(int) result.setPageSize(int)</pre>	<p>Limit the results to the first <code>int</code> number of rows at the top, after grouping and ordering. These query builder API methods provide similar functionality in a portable way.</p>

Unsupported SQL features in query builder APIs

Query builder APIs provide no equivalent for these features of SQL SELECT statements. ClaimCenter never generates SQL statements that contain these keywords or clauses.

Keyword or clause in SQL	Meaning in SQL	Why the APIs do not support it
FROM <i>table1</i> , <i>table2</i> [, ...]	Declares the tables in a join query, beginning with the primary source of information on the left and proceeding with dependent sources of information towards the right.	<ul style="list-style-type: none"> This natural way of specifying a join can produce inappropriate results if the WHERE clause is not written correctly. Relational query performance often suffers when SQL queries include this syntax. You can specify only natural inner joins with this SQL syntax.

Keyword or clause in SQL	Meaning in SQL	Why the APIs do not support it
EXCEPT	Removes items fetched by a query that are in the results fetched by a second query.	This SQL feature is seldom used.
INTERSECT	Reduce items fetched by two separate queries to those items in both results only.	Query intersection often causes a performance problem. A better choice is to use a more efficient query type. For example, if both sides of the INTERSECT clause query the same table, use a single query. Use an AND operator to combine the restrictions from both sides of the INTERSECT to restrict the result.

See also

- “Building a simple query” on page 748
- “Making an inner join with the foreign key on the right” on page 770
- “Working with row queries” on page 780
- “Joining a related entity to a query” on page 767
- “Restricting a query with predicates on fields” on page 751
- “Ordering results” on page 794
- “Applying a database function to a column” on page 782
- “Accessing the first item in a result” on page 798
- “Setting the page size for prefetching query results” on page 804
- “Determining if a result will return too many items” on page 798

The processing cycle of a query

Two types of objects drive the processing cycle of ClaimCenter queries:

A query object

Specifies which ClaimCenter entity instances to fetch from the application database

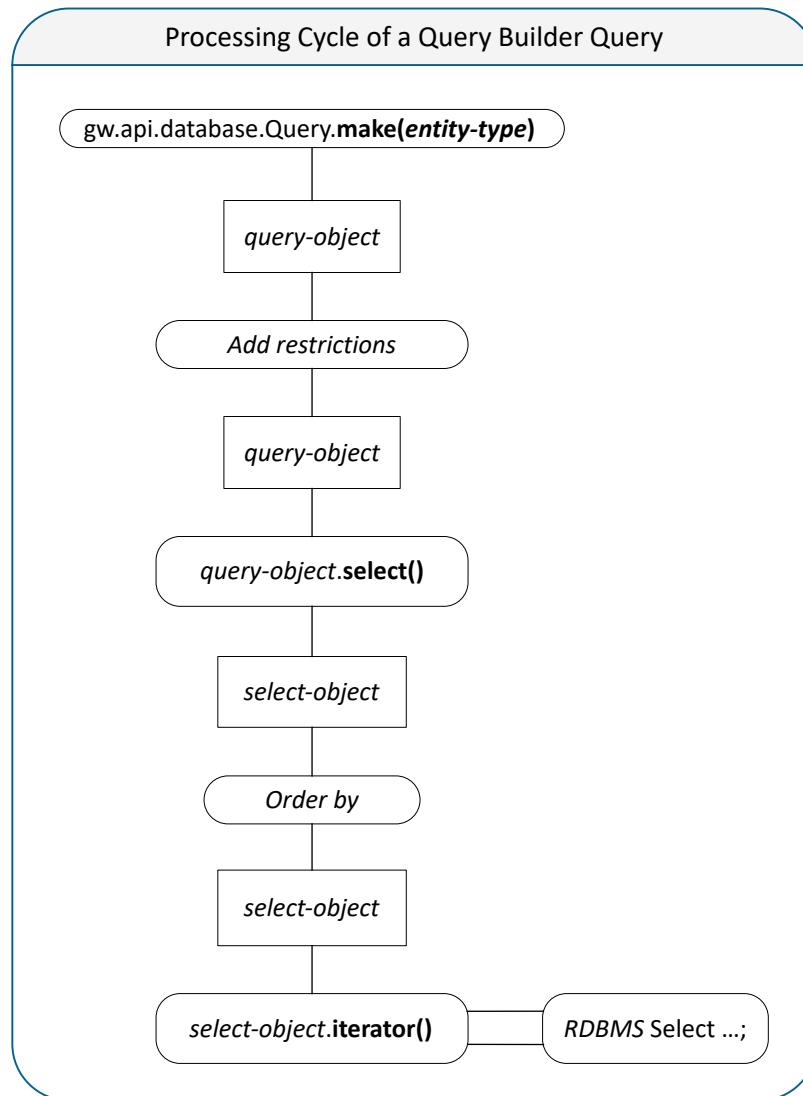
A select object

Specifies how to order and group selected entity instances

The processing cycle of a ClaimCenter query follows these high-level steps:

1. Invoke the static `gw.api.database.Query.make(EntityType)` method, which creates a query object.
2. Refine your query object with restrictions.
3. Invoke the `select` method on your query object, which creates a select object.
4. Refine your select object by ordering the selected items.
5. Iterate your select object with methods that the `java.lang.Iterable<T>` interface defines or with a `for` loop.

The following diagram illustrates the processing cycle of a query.



The query builder APIs send queries to the application database when your code accesses information from the result set, not when your code calls the `select` method. Although your code seems to order results after fetching data, the query is not performed until you start to access the result set. The application database orders the results while fetching data. Any action that you take on result objects to return information, such as getting result counts or starting to iterate the result, triggers query execution in the application database.

Building a simple query

Consider a simple query in SQL that returns information from a single database table. The SQL `SELECT` statement specifies the table. Without further restrictions, the database returns all rows and columns of information from the table.

For example, you submit the following SQL statement to a relational database.

```
SELECT * FROM addresses;
```

In response, the relational database returns a result set that contains fetched information. A *result set* is like a database table, with columns and rows, that contains the information that you specified with the `SELECT` statement. In response to the preceding SQL example code, the relational database returns a result set that has the same columns and rows as the `addresses` table.

The following Gosu code constructs and executes a functionally equivalent query to the preceding SQL example.

```
uses gw.api.database.Query      // Import the query builder APIs.  
var query = Query.make(Address)  
var select = query.select()  
var result = select.iterator() // Execute the query and access the results with an iterator.
```

IMPORTANT: Using the Query Builder API to create a query is different in Java and Gosu. In Java, you must use the `gw.api.database.Queries.createQuery` method rather than `gw.api.database.Query.make`.

The following Java code is equivalent to the preceding Gosu code.

```
import entity.Address;  
import gw.api.database.IQueryBeanResult;  
import gw.api.database.Queries;  
import gw.api.database.Query;  
import java.util.Iterator;  
...  
Query<Address> query = Queries.createQuery(Address.TYPE);  
IQueryBeanResult<Address> select = query.select();  
Iterator result = select.iterator();
```

In response, the ClaimCenter application database returns a result object that contains fetched entity instances. A *result object* is like a Gosu collection that contains entity instances of the type that you specified with the `make` method and that meets any restrictions that you added. Calling the `iterator` method in the preceding Gosu code causes the application database to fetch all `Address` instances from the application database into the result object.

The query builder API can use a view entity as the primary entity type in the same way as a standard entity. A view entity can provide straightforward access to a subset of columns on primary and related entities.

See also

- [Configuration Guide](#)

Restricting the results of a simple query

Generally, you want to restrict the information that queries return from a database instead of selecting all the information that the database contains. With SQL, the `WHERE` clause lets you specify Boolean expressions that data must satisfy to be included in the result.

For example, you submit the following SQL statement to a relational database.

```
SELECT * FROM addresses  
WHERE city = "Chicago";
```

In response, the relational database returns a result set with addresses only in the city of Chicago. In the preceding SQL example, the Boolean expression applies the predicate `= "Chicago"` to the column `city`. The expression asserts that addresses in the result set have “Chicago” as the city.

The following Gosu code constructs and executes a functionally equivalent query to the preceding SQL example.

```
uses gw.api.database.Query  
var query = Query.make(Address)  
query.compare(Address#City, Equals, "Chicago")  
var select = query.select()  
var result = select.iterator() // Execute the query and return an iterator to access the result.
```

In response to the preceding Gosu code, the application database fetches all `Address` instances from the application database that are in the city of Chicago.

Ordering the results of a simple query

Relational databases can return result sets with rows in a seemingly random order. With SQL, the ORDER BY clause lets you specify how the database sorts fetched data. The following SQL statement sorts the result set on the postal codes of the addresses.

```
SELECT * FROM addresses
  WHERE city = 'Chicago'
  ORDER BY postal_code;
```

The following Gosu code sorts the instances in the result object in the same way as the preceding SQL example.

```
uses gw.api.database.Query
uses gw.api.database.QuerySelectColumns // Import the class that defines a column
uses gw.api.path.Paths // Import the class that builds a path to a column

var query = Query.make(Address)
query.compare(Address#City, Equals, "Chicago")
var select = query.select()
// Specify to sort the result by postal code.
select.orderBy(QuerySelectColumns.path(Paths.make(Address#PostalCode)))
```

See also

- “Ordering results” on page 794

Accessing the results of a simple query

You use queries to fetch information from a database, and then work with the results. You embed SQL queries in your application code so you can bind application data to the SQL queries and submit them programmatically to the ClaimCenter relational database. Result sets that relational databases return provide a programmatic cursor to let you access each result row sequentially.

You use the query builder APIs to embed queries naturally in your Gosu application code. You bind application data to your queries and submit the queries programmatically with standard Gosu syntax. ClaimCenter application databases return select objects that implement the `java.lang.Iterable<T>` interface to let you access each entity instance in the result sequentially.

For example, the following Gosu code creates a query of the `Address` entity instances in the application database. The query binds the constant “Chicago” to the `compare` predicate on the property `City`, which restricts the results.

```
uses gw.api.database.Query
uses gw.api.database.QuerySelectColumns
uses gw.api.path.Paths

// Specify what you want to select.
var query = Query.make(Address)
query.compare(Address#City, Equals, "Chicago")

// Specify how you want the result returned.
var select = query.select()
select.orderBy(QuerySelectColumns.path(Paths.make(Address#PostalCode)))

// Fetch the data with a for loop and print it.
for (address in select) {
    print (address.AddressLine1 + ", " + address.City + ", " + address.PostalCode)
}
```

When the preceding code begins the `for` loop, the application database fetches `Address` instances in the city of Chicago and sorts them by postal code. The `for` loop passes addresses in the result one at a time to the statements inside the `for` block. The `for` block prints each address on a separate line.

Alternatively, the following Gosu code uses an iterator and a `while` loop to fetch matching `Address` instances in the order specified.

```
uses gw.api.database.Query
uses gw.api.database.QuerySelectColumns
uses gw.api.path.Paths

// Specify what you want to select.
var query = Query.make(Address)
```

```
query.compare(Address#City, Equals, "Chicago")
// Specify how you want the result returned.
var select = query.select()
select.orderBy(QuerySelectColumns.path(Paths.make(Address#PostalCode)))

// Fetch the data with a while loop and print it.
var result = select.iterator()
while (result.hasNext()) {
    var address = result.next()
    print (address.AddressLine1 + ", " + address.City + ", " + address.PostalCode)
}
```

There are no consequential functional or performance differences between iterating results with a `for` loop and iterating results with an iterator.

Restricting a query with predicates on fields

In most cases, you do not need to process all rows in a table. Typically, you need to restrict the set of rows, for example, by date or geographic location.

A *predicate* is a condition that selects a subset of values from a larger set. Predicates are independent of the sets to which you apply them. You can create predicates to apply to multiple sets of values to select subsets of those values. For example, the expression “is male” is a predicate that comprises a comparison operator, “is,” and a comparison value, “male.” You can apply this predicate to different sets of people. If you apply the predicate “is male” to the set of people in your family, the predicate selects all the male members of your family. If you apply the predicate to the set of people in your work group, “is male” selects a different, possibly overlapping subset of male people.

SQL SELECT statements provide WHERE and HAVING clauses to apply predicates to the sets of values in specific database columns. The query builder APIs provide methods on query objects to apply predicates to the sets of values in specific entity fields.

See also

- “Restricting query results by using fields on joined entities” on page 776
- “Restricting query results with fields on primary and joined entities” on page 778

Using a comparison predicate with a character field

The result of comparing the value of a character field with another character value differs depending on the language, search collation strength, and database that your ClaimCenter uses. For example, case-insensitive comparisons produce the same results as case-sensitive comparisons if ClaimCenter has a linguistic search strength of primary.

ClaimCenter converts character literals that you specify in query builder predicates to bind variables in the SQL statement that ClaimCenter sends to the relational database.

See also

- Globalization Guide*

Case-sensitive comparison of a character field

In SQL, you apply comparison predicates to column values in the WHERE clause. For example, the following SQL statement applies the predicate = “Acme Rentals” to values in the name column of the companies table to select the company “Acme Rentals.”

```
SELECT * from companies
WHERE name = "Acme Rentals";
```

With the query builder APIs, you apply predicates to entity fields by using methods on the query object. The following Gosu code applies the `compare` method to values in the Name field of Company entity instances to select the company “Acme Rentals.”

```
uses gw.api.database.Query
```

```
// Query the Company instances for a specific company.
var query = Query.make(Company)
query.compare(Company#Name, Equals, "Acme Rentals")

// Fetch the data and print it.
var result = query.select()
for (company in result) {
    print (company.Name)
}
```

In the result set, the values in the character field exactly match the comparison value that you provide to the `compare` method, including the case of the comparison value. For example, the preceding query builder code selects only “Acme Rentals,” but not “ACME Rentals.”

Note: In configurations that use primary or secondary linguistic sort strength, the preceding query builder code does not perform a case-sensitive comparison. If either of your ClaimCenter or the relational database that your ClaimCenter uses have these configurations, the result includes both “Acme Rentals” and “ACME Rentals.”

See also

- “Comparing a column value with a literal value” on page 761
- “Comparing a typekey column value with a typekey literal” on page 762
- *Globalization Guide*

Case-insensitive comparison of a character field

The case of letters can cause a problem when you compare character values. You might not know the case of individual characters in the field values that you want to select. For example, you want to select the company named “Acme Rentals.” There is only one instance of the `Company` entity type with that name. However, you do not know whether the company name in the database is “Acme Rentals,” “ACME RENTALS,” or even “acme rentals.”

SQL WHERE clauses let you apply case-insensitive comparisons with functions that convert values in columns to upper or lower case before making a comparison. The following example SQL statement converts values in the `name` column to lower case before applying the predicate = “`acme rentals`”.

```
SELECT * from companies
WHERE LCASE(name) = "acme rentals";
```

The query builder APIs provide the `compareIgnoreCase` method for case-insensitive comparisons, as the following Gosu code shows.

```
uses gw.api.database.Query

var query = Query.make(Company)
query.compareIgnoreCase(Company#Name, Equals, "Acme Rentals")

// Fetch the data print it.
var result = query.select()
for (company in result) {
    print (company.Name)
}
```

IMPORTANT: Your data model definition must specify the column with the attribute `supportsLinguisticSearch` set to `true`. Otherwise, query performance suffers if you include the column in a `compareIgnoreCase` predicate method.

See also

- *Globalization Guide*

Comparison of a character field to a range of values

In SQL, you use the `BETWEEN` keyword to apply comparison predicates to an inclusive range of column values in the `WHERE` clause. For example, the following SQL statement applies the predicate `BETWEEN "Bank" AND "Business"` to

values in the name column of the companies table. The query returns all companies with names between the values “Bank” and “Business,” including any company that has the name “Bank” or the name “Business.”

```
SELECT * from companies
WHERE name BETWEEN "Bank" AND "Business";
```

With the query builder APIs, you use the `between` method to apply a range predicate to entity fields. The following Gosu code applies the `between` method to values in the Name field of Company entity instances to select companies with names between the values “Bank” and “Business.”

```
uses gw.api.database.Query

// Query the Company instances for a range of company names.
var query = Query.make(Company)
query.between(Company#Name, "Bank", "Business")

// Fetch the data and print it.
var result = query.select()
for (company in result) {
    print (company.Name)
}
```

The `between` method performs a case-sensitive comparison of the values in the character field to the comparison values. For example, the preceding query builder code selects only “Building Renovators,” but not “building renovators.”

Note: In configurations that use primary or secondary linguistic sort strength, the preceding query builder code does not perform a case-sensitive comparison. If either of your ClaimCenter or the relational database that your ClaimCenter uses have these configurations, the result includes both “Building Renovators” and “building renovators.”

To perform a case-insensitive comparison on a range of values for a character field, use the `compareIgnoreCase` method, as the following Gosu code shows.

```
uses gw.api.database.Query

// Query the Company instances for a range of company names.
var query = Query.make(Company)
query.compareIgnoreCase(Company#Name, GreaterThanOrEquals, "bank")
query.compareIgnoreCase(Company#Name, LessThanOrEquals, "business")

// Fetch the data and print it.
var result = query.select()
for (company in result) {
    print (company.Name)
}
```

See also

- “Handling of null values in a range comparison” on page 758

Partial comparison from the beginning of a character field

Sometimes you need to make a partial comparison that matches the beginning of a characters field, instead of matching the entire field. For example, you want to select a company named “Acme”, but you do not know whether the full name is “Acme Company”, “Acme, Inc.”, or even “Acme”.

In SQL, you apply a partial comparison predicate with the `LIKE` operator in the `WHERE` clause. The following SQL statement applies the predicate `LIKE "Acme%"` to values in the name column of the companies table. The percent sign (%) is an SQL wildcard symbol that matches zero or more characters.

```
SELECT * from companies
WHERE name LIKE "Acme%";
```

The query result contains companies with names that begin with “Acme”.

With the query builder APIs, you apply partial comparison predicates that match from the beginnings of character fields by using the `startsWith` method on the query object. Test the use of the `startsWith` method in a realistic

environment. Using the `startsWith` method as the most restrictive predicate on a query can cause a delay on the user interface.

The following Gosu code applies the `startsWith` predicate method to values in the `Name` field on `Company` entity instances.

```
uses gw.api.database.Query

// Query the Company instances for a specific company.
var queryCompany = Query.make(Company)
queryCompany.startsWith(Company#Name, "Acme", false)

// Fetch the data with a for loop.
var result = queryCompany.select()
for (company in result) {
    print (company.Name)
}
```

The sample code prints the names of companies that begin with “Acme”.

Note: The value `false` as the third argument of the `startsWith` method specifies case-sensitive matching of values. In configurations that use primary or secondary linguistic sort strength, the preceding query builder code does not perform a case-sensitive comparison. If either your ClaimCenter application or the relational database that your ClaimCenter uses have these configurations, the result includes the names of companies that begin with both “Acme” and “ACME.”

See also

- “Case-insensitive partial comparisons of a character field” on page 755
- *Globalization Guide*

Partial comparison of any portion of a character field

Sometimes you need to make a partial comparison that matches any portion of a character field, not just from the beginning of the field. For example, you want to select an activity that is a review, but you do not know the type of review. In this case, you want to search for “Review” anywhere within the field for the subject of the activity.

In SQL, you apply partial comparison predicates with the `LIKE` operator in the `WHERE` clause. The following SQL statement applies the predicate `LIKE "%Review%"` to values in the `Subject` column of the `cc_activity` table. The percent sign (%) is an SQL wildcard symbol that matches zero or more characters. The query also restricts the set of activities to those that have a particular assigned user. Using the `LIKE` operator without another, more restrictive predicate causes the database to do a full table scan because no index is available to the query.

```
SELECT * FROM cc_activity
INNER JOIN cc_user
    ON cc_user.ID = cc_activity.AssignedUserID
WHERE cc_activity.Subject LIKE "%Review%"
    AND cc_user.PublicID = "cc:8";
```

The query result contains activities with a subject that includes “Review” and that have a particular assigned user.

The following query uses the `LIKE` operator in the `WHERE` clause and causes a full table scan because there is no other restriction on the primary table:

```
SELECT * FROM cc_activity
WHERE cc_activity.Subject LIKE "%Review%";
```

With the query builder APIs, you apply partial comparison predicates that match any portion of character fields by using the `contains` method on the query object. Test the use of the `contains` method in a realistic environment. Using the `contains` method as the most restrictive predicate on a query causes a full-table scan in the database because the query cannot use an index.

WARNING: For a query on a large table, using `contains` as the most restrictive predicate can cause an unacceptable delay to the user interface.

The following Gosu code applies the `contains` predicate method to values in the `Subject` field on the `Activity` instance for which a `User` instance is the assigned user.

```
uses gw.api.database.Query

// Query the Activity instances for a particular user based on subject line.
var queryActivity = Query.make(Activity)

// Join User as the dependent entity to the primary entity Activity.
queryActivity.join(Activity#AssignedUser).compare(User#PublicID, Equals, "demo_sample:1")

// Add a predicate on the primary entity.
// The value "false" means "case-sensitive."
queryActivity.contains(Activity#Subject, "Review", false)

// Fetch the data with a for loop.
for (activity in queryActivity.select()) {
    print("Assigned to " + activity.AssignedUser + ": " + activity.Subject)
}
```

This Gosu code prints the assigned user and the subject of the activity where the subject of the activity contains “Review” and the assigned user has a particular public ID.

Note: The value `false` as the third argument of the `contains` method specifies case-sensitive matching of values. In configurations that use primary or secondary linguistic sort strength, the preceding query builder code does not perform a case-sensitive comparison. If either your ClaimCenter application or the relational database that your ClaimCenter uses have these configurations, the result includes the subjects of activities that contain both “Review” and “review.”

See also

- “Case-insensitive partial comparisons of a character field” on page 755

Case-insensitive partial comparisons of a character field

Sometimes you need to make partial comparisons in a case-insensitive way. In SQL, you apply case-insensitive partial comparison predicates with functions that convert the column values to upper or lower case before making the comparison. The following SQL statement converts the values in `Subject` to lower case before comparing them to `review`.

```
SELECT * FROM cc_activity
INNER JOIN cc_user
ON cc_user.ID = cc_activity.AssignedUserID
WHERE LOWER(cc_activity.Subject) LIKE "%review%"
AND cc_user.PublicID = "cc:8";
```

With the query builder APIs, you use the third parameter of the `startsWith` and `contains` methods to specify whether to make the comparison in a case-insensitive way.

- `true` – Case-insensitive comparisons
- `false` – Case-sensitive comparisons

IMPORTANT: In configurations using primary or secondary linguistic sort strength, query builder code does not perform a case-sensitive comparison even if you use a value of `false` for the third parameter.

The following Gosu code applies the `contains` predicate method to values in the `Subject` field on the `Activity` instance for which a `User` instance is the assigned user. The code requests a case-insensitive comparison with the value `true` in the third parameter.

```
uses gw.api.database.Query

// Query the Activity instances for a particular user based on subject line.
var queryActivity = Query.make(Activity)

// Join User as the dependent entity to the primary entity Activity.
queryActivity.join(Activity#AssignedUser).compare(User#PublicID, Equals, "demo_sample:1")

// Add a predicate on the primary entity.
// The value "true" means "case-insensitive."
queryActivity.contains(Activity#Subject, "Review", true)
```

```
// Fetch the data with a for loop.
for (activity in queryActivity.select()) {
    print("Assigned to " + activity.AssignedUser + ": " + activity.Subject)
}
```

If you choose case-insensitive partial comparisons, Gosu generates an SQL function that depends on your ClaimCenter and database configuration to implement the comparison predicate. If the data model definition of the column sets the `supportsLinguisticSearch` attribute to `true`, Gosu uses the denormalized version of the column instead.

See also

- *Globalization Guide*

SQL wildcard symbols for partial character comparisons

SQL supports the following wildcard symbols in the `LIKE` operator:

% (percent sign)

Represents zero, one, or more characters

_ (underscore)

Represents exactly one character

The query builder API does not support the use of SQL wildcard symbols in the `startsWith` or `contains` predicate methods. If you include wildcard symbols in the search-term parameter of either method, the query builder methods escape their special meaning by preceding them with a backslash (\).

Using a comparison predicate with a date and time field

Different brands of database provide different functions to work with timestamp columns in SQL query statements. The query builder APIs offer the following database functions for working with timestamp columns on any supported database:

- `DateDiff` – Computes the interval between two date and time fields
- `DatePart` – Retrieves the parts of a date-and-time field
- `DateFromTimestamp` – Retrieves the date part of a date-and-time field

Comparing the interval between two date and time fields

Use the static `DateDiff` method of the `DBFunction` class to compute the interval between two `java.util.Date` fields. The `DateDiff` method takes two `ColumnRef` parameters to timestamp fields in the database: a starting timestamp and an ending timestamp. The `DateDiff` method returns the interval between the two fields.

```
DateDiff(dateDiffPart : DateDiffPart, startDate : ColumnRef, endDate : ColumnRef) : DBFunction
```

You use the initial parameter of the method, `dateDiffPart`, to specify the unit of measure for the result. You can specify `DAYS`, `HOURS`, `SECONDS`, or `MILLISECONDS`. If `endDate` precedes the `startDate`, the method returns a negative value for the interval, instead of a positive value.

Example of the `DateDiff` method

The following Gosu code uses the `DateDiff` method to compute the interval between the assigned date and the due date on an activity. The query builder code uses the returned interval in a comparison predicate to select activities with due dates less than 15 days from their assignment dates.

```
uses gw.api.database.Query
uses gw.api.database.DBFunction
uses gw.api.database.QuerySelectColumns
uses gw.api.path.Paths

var query = Query.make(Activity)
// Query for activities with due dates less than 15 days from the assigned date.
query.compare(DBFunction.DateDiff(DAYS,
```

```

query.getColumnRef("AssignmentDate"), query.getColumnRef("EndDate")), LessThan, 15)

// Order the result by assignment date.
for (activity in (query.select().orderBy(
    QuerySelectColumns.path(Paths.make(Activity#AssignmentDate)))))) {
    print("Assigned on " + activity.AssignmentDate + ": " + activity.DisplayName)
}

```

Effect of daylight saving time on the return value of the DateDiff method

The `DateDiff` method does not adjust for daylight saving, or summer, time. For example, consider a locale in which daylight saving time ends on the first Sunday in November. A call to the `DateDiff` method requests the number of hours between the `java.util.Date` values `2017-11-04 12:00` and `2017-11-05 12:00`. The method returns an interval of 24 hours, even though 25 hours separate the two in solar time.

Comparing parts of a date and time field

Use the static `DatePart` method of the `DBFunction` class to extract a portion of a `java.util.Date` field to use in a comparison predicate. The `DatePart` method takes two parameters. The first parameter specifies the part of the date and time you want to extract, and the second parameter specifies the field from which to extract the part. The `DatePart` method returns the extracted part as a `java.lang.Number`.

```
DatePart(datePart : DatePart, date : ColumnRef) : DBFunction
```

For the `datePart` parameter, you can specify `HOUR`, `MINUTE`, `SECOND`, `DAY_OF_WEEK`, `DAY_OF_MONTH`, `MONTH`, or `YEAR`. When you specify `DAY_OF_WEEK` as the part to extract, the first day of the week is Monday.

The following Gosu codes uses the `DatePart` method to extract the day of the month from the due date on activities. The query builder codes uses the returned numeric value in a comparison predicate to select activities that are due on the 15th of any month.

```

uses gw.api.database.Query
uses gw.api.database.DBFunction
uses gw.api.database.QuerySelectColumns
uses gw.api.path.Paths

var query = Query.make(Activity)
// Query for activities with due dates that fall on the 15th of any month.
query.compare(DBFunction.DatePart(DAY_OF_MONTH, query.getColumnRef("endDate")), Equals, 15)

// Order the result by assignment date and iterate the items fetched.
for (activity in (query.select().orderBy(
    QuerySelectColumns.path(Paths.make(Activity#AssignmentDate)))))) {
    print("Assigned on " + activity.AssignmentDate + ": " + activity.DisplayName)
}

```

Comparing the date part of a date and time field

Use the static `DateFromTimestamp` method of the `DBFunction` class to extract the date portion of a `java.util.Date` field to use in a comparison predicate. The `DateFromTimestamp` method takes a single parameter, a column reference to a `java.util.Date` field. The return value is a `java.util.Date` with only the date portion specified.

```
DateFromTimestamp(timestamp : ColumnRef) : DBFunction
```

The following Gosu code uses the `DateFromTimestamp` method to extract the date from the creation timestamp on activities. The query builder code uses the returned date in a comparison predicate to select activities that were created some time during the current day.

```

uses gw.api.database.Query
uses gw.api.database.DBFunction
uses gw.api.database.QuerySelectColumns
uses gw.api.path.Paths
uses gw.api.util.DateUtil

// Make a query of Address instances.
var query = Query.make(Address)

// Query for addresses created today.
query.compare(DBFunction.DateFromTimestamp(query.getColumnRef("CreateTime")),
    Equals, DateUtil.currentDate())

```

```
// Order the result by creation date and iterate the items fetched.
for (address in query.select().orderBy(QuerySelectColumns.path(Paths.make(Address#CreateTime)))) {
    print(address.DisplayName + ": " + address.CreateTime)
}
```

Comparison of a date and time field to a range of values

In SQL, you use the BETWEEN keyword to apply comparison predicates to an inclusive range of column values in the WHERE clause. With the query builder APIs, you use the between method to apply a range predicate to entity fields. The following Gosu code applies the between method to values in the creation timestamp field of Address entity instances to select addresses that were created in the previous month.

```
uses gw.api.database.Query
uses gw.api.database.DBFunction
uses gw.api.database.QuerySelectColumns
uses gw.api.path.Paths
uses gw.api.util.DateUtil

// Query the Company instances for a range of creation dates.
var firstDayOfCurrentMonth = DateUtil.currentDate().FirstDayOfMonth
var previousMonthStart = DateUtil.addMonths(firstDayOfCurrentMonth, -1)
var previousMonthEnd = DateUtil.addDays(firstDayOfCurrentMonth, -1)

// Make a query of Address instances.
var query = Query.make(Address)
// Query for addresses created in the previous month.
query.between(DBFunction.DateFromTimestamp(query.getColumnRef("CreateTime")),
    previousMonthStart, previousMonthEnd)

// Order the result by creation date and iterate the items fetched.
for (address in query.select().orderBy(QuerySelectColumns.path(Paths.make(Address#CreateTime)))) {
    print(address.DisplayName + ": " + address.CreateTime)
}
```

See also

- “Handling of null values in a range comparison” on page 758

Using a comparison predicate with a null value

In a relational database, you can define columns that allow null values or that require every row to have a value. The equivalent in a ClaimCenter application is to define entity properties that allow null values or that require every instance to have a value.

Selecting instances based on null or non-null values

Use the compare method with the Equals or NotEquals operator to select entity instances based on null or non-null values. The following Gosu code returns all Person instances where the birthday is unknown.

```
uses gw.api.database.Query

var query = Query.make(Person)
query.compare(Person#DateOfBirth, Equals, null)
```

The following Gosu code returns all Address instances where the first address line is known.

```
uses gw.api.database.Query

var query = Query.make(Address)
query.compare(Address#AddressLine1, NotEquals, null)
```

Handling of null values in a range comparison

If one of the two comparison values that you pass to the between method is null, Gosu performs a simple comparison that is equivalent to calling the compare method. If the first comparison value is null, Gosu performs a less-than-or-

equal comparison on the second value. If the second comparison value is `null`, Gosu performs a greater-than-or-equal comparison on the first value. For example, the following two statements are equivalent:

```
query.between(Company#Name, null, "Business")
query.compare(Company#Name, LessThanOrEquals, "Business")
```

How null values get in the database

Null values get in the database only for entity properties that the *Data Dictionary* does not define as non-null. To assign `null` values to entity instance properties, use the special Gosu value `null`. The following Gosu code sets an `int` property and a `java.util.Date` property to `null` on a new entity instance.

```
var aPerson = new Person()
aPerson.DateOfBirth = null      // Set a java.util.Date to null in the database
aPerson.NumDependents = null    // Set an int to null in the database
```

After the bundle with the new `Person` instance commits, its `DateOfBirth` and `NumDependents` properties are `null` in the database.

Blank and empty strings become null in the database

To assign `null` values to `String` properties, use the special Gosu value `null` or the empty string (""). If you set the property of an entity instance to a blank or empty string, ClaimCenter coerces the value to `null` when it commits the instance to the database.

The following Gosu code sets three `String` properties to different values.

```
var anAddress = new Address()
anAddress.AddressLine1 = " "      // Sets a String to null in the database
anAddress.AddressLine2 = ""       // Sets a String to null in the database
anAddress.AddressLine3 = null     // Sets a String to null in the database
```

After the bundle with the new `Address` instance commits, all three address lines are `null` in the database. Before ClaimCenter commits `String` values to the database, it trims leading and trailing spaces. If the result is the empty string, ClaimCenter coerces the value to `null`.

Note that for non-null `String` properties, you must provide at least one non-whitespace character. You cannot work around a non-null requirement by setting the property to a blank or empty string.

Controlling whether ClaimCenter trims whitespace before committing string properties

You can control whether ClaimCenter trims whitespace before committing a `String` property to the database with the `trimwhitespace` column parameter in the data model definition of the `String` column. Columns that you define as `type="varchar"` trim leading and trailing spaces by default.

To prevent ClaimCenter from trimming whitespace before committing a `String` property to the database, add the `trimwhitespace` column parameter in the column definition, and set the parameter to `false`. The XML text of a column definition that does not trim whitespace looks like the following:

```
<column
  desc="Primary email address associated with the contact."
  name="EmailAddress1"
  type="varchar">
  <columnParam name="size" value="60"/>
  <columnParam name="trimwhitespace" value="false"/>
</column>
```

The parameter controls only whether ClaimCenter trims leading and trailing spaces. You cannot configure whether ClaimCenter coerces an empty string to `null`.

See also

- [Gosu Reference Guide](#)

Using set inclusion and exclusion predicates

In SQL, you can embed an SQL query in-line within another query. A query that is embedded inside another query is called a *subselect* or a *subquery*. The SQL query language does not provide a keyword to create a subselect. The structure of your SQL query creates a subselect, as shown in the following SQL syntax, which creates a subselect to provide a set of values for an IN predicate.

```
SELECT column_name1 FROM table_name
  WHERE column_name1
    IN (SELECT column_name2 FROM table_name2
      WHERE column_name2 LIKE '%some_value%');
```

The following example uses a subselect to find users that have created notes of the specified topic types:

```
SELECT ID FROM cc_user
  WHERE ID
    IN (SELECT AuthorID FROM cc_note
      WHERE Topic IN (1, 10006));
```

The following Gosu code constructs and executes a functionally equivalent query to the preceding SQL example.

```
uses gw.api.database.Query
uses gw.api.database.InOperation

var outerQuery = Query.make(User) // Returns User Query
var innerQuery = Query.make(Note) // Returns Note Query

// Filter the inner query
innerQuery.compareIn(Note#Topic, {NoteTopicType.TC_GENERAL, NoteTopicType.TC_LEGAL})
// Filter the outer query by using a subselect
outerQuery.subselect(User#ID, InOperation.CompareIn, innerQuery, Note#Author)

// Alternatively, use simpler syntax if you do not need to filter the inner query
// queryParent.subselect(User#ID, InOperation.CompareIn, Note#Author)

var result = outerQuery.select() // Returns User query Object result
for (user in result) {           // Execute the query and iterate the results
    print(user.DisplayName)
}
```

The `subselect` method is overloaded with many method signatures for different purposes. The preceding Gosu code shows two of these method signatures.

The query builder APIs generate an SQL IN clause for the `CompareIn` method. The query builder APIs generate an SQL NOT EXISTS clause for the `CompareNotIn` method. These methods do not accept a null value in the list of values. A run-time error occurs if you provide a null value in set of comparison values.

Comparing column values with each other

Some predicate methods have signatures that let you compare a column value on an entity instance with another column value. These methods use a property reference to access the first column and a column reference to access the second column. Use the `getColumnRef` method on query, table, and restriction objects to obtain a column reference.

Note: The `ColumnRef` object that the `getColumnRef` method returns does not implement the `IQueryBuilder` interface. You cannot chain the `getColumnRef` method with other query builder methods.

For example, you need to query the database for contacts where the primary and secondary email addresses differ. The following SQL statement compares the two email address columns on a contact row with each other.

```
SELECT * FROM cc_contact
  WHERE EmailAddress1 <> EmailAddress2;
```

The following Gosu code constructs and executes a functionally equivalent query to the preceding SQL statement. This Gosu code uses the `getColumnRef` method to retrieve the second email address property.

```
uses gw.api.database.Query

var query = Query.make(Contact) // Query for contacts where email 1 and email 2 do not match.
query.compare(Contact#EmailAddress1, NotEquals, query.getColumnRef("EmailAddress2"))
```

```
var result = query.select()  
  
for (contact in result) { // Execute the query and iterate the results  
    print("public ID '" + contact.PublicID + " with name " + contact.DisplayName  
         + " and emails " + contact.EmailAddress1 + "," + contact.EmailAddress2)  
}
```

See also

- “Chaining query builder methods” on page 806

Comparing a column value with a literal value

You can pass a Gosu literal as a comparison value to predicate methods. ClaimCenter generates and sends SQL that treats the literal value as an SQL query parameter to the relational database. Use the `DBFunction.Constant` static method to specify literal values to treat as SQL literals in the SQL query that ClaimCenter submits to the relational database.

If your query builder code uses the same Gosu literal in all invocations, replace the Gosu literal with a call to the `DBFunction.Constant` method. Using this method to coerce a Gosu literal to an SQL literal can improve query performance. To improve query performance, you must compare the literal value to a column that is in an index, and one of the following:

- Your query builder code always passes the same literal value.
- Your literal value is either sparse or predominant in the set of values in the database column.

IMPORTANT: Use the `DBFunction.Constant` method with caution. If you are unsure, consult your database administrator for guidance.

Differences between Gosu literals and database constants

Gosu literal in a query example 1

The following Gosu code uses the Gosu literal "Los Angeles" to select addresses from the application database.

```
uses gw.api.database.Query  
  
// Query for addresses by using a Gosu literal to select addresses in Los Angeles.  
var query = Query.make(Address)  
query.compare(Address#City, Equals, "Los Angeles")  
  
var result = query.select()  
  
for (address in result) {  
    print("public ID " + address.PublicID + " with city " + address.City)  
}
```

The code specifies a literal value for the predicate comparison, but the SQL query that the code generates uses the value "Los Angeles" as a prepared parameter.

SQL literal in a query example 2

In contrast, the following Gosu code uses the `DBFunction.Constant` method to force the generated SQL query to use the Gosu string literal "Los Angeles" as an SQL literal.

```
uses gw.api.database.Query  
uses gw.api.database.DBFunction  
  
// Query for addresses by using an SQL literal to select addresses in Los Angeles.  
var query = Query.make(Address) // Query for addresses.  
query.compare(Address#City, Equals, DBFunction.Constant("Los Angeles"))  
  
var result = query.select()  
  
for (address in result) {  
    print("public ID " + address.PublicID + " with city " + address.City)  
}
```

Constant method signatures

The `DBFunction.Constant` method has these signatures:

```
Constant(value String) : DBFunction
Constant(value java.lang.Number) : DBFunction
Constant(dataType IDataType, value Object) : DBFunction
```

Comparing a typekey column value with a typekey literal

Often you want to select data based on the values of typekey fields. A *typekey field* takes its values from a specific typelist. A *typelist* contains a set of codes and related display values that appear in the drop-down lists of the ClaimCenter application.

Gosu provides *typekey literals* with which you specify typelist codes in your programming code. Gosu creates typekey literals at compile time by prefixing typelist codes with `TC_` and converting code values to upper case. For example, the following Gosu code specifies the typekey code for open in the `ActivityStatus` typelist.

```
typekey.ActivityStatus.TC_OPEN
```

The `Address` entity type has a typekey field named `State` that takes its values from the `State` typelist. The following sample code uses the typekey literal for California to select and print all addresses in California.

```
uses gw.api.database.Query
// Query the database for addresses.
var query = Query.make(Address)

// Restrict the query to addresses in the state of California.
query.compare(Address#State, Equals, typekey.State.TC_CA)

var result = query.select()

// Iterate and print the result.
for (address in result) {
    print(address.AddressLine1 + ", " + address.City + ", " + address.State)
}
```

Use code completion in Studio to build complete object path expressions for typelist literals. To begin, type `typekey.`, and then work your way through the typelist names to the typekey code that you need.

See also

- [Gosu Reference Guide](#)
- [Configuration Guide](#)

Using a comparison predicate with spatial data

Often, you want to compare one geographical location to another geographical location with a distance calculation. In particular, you desire to select entity instances having a property that pinpoints a location that is within a given distance of a reference location. To do this, use the appropriate `withinDistance` method in connection with an entity query.

The following code examples demonstrate how to use two varieties of the overloaded `withinDistance` method. The first code example uses the `withinDistance` method in connection with a simple entity query. This query selects those `Address` entity instances with a `SpatialPoint` property that is within 10 miles of a `SpatialPoint` constant representing San Mateo. The first code example is as follows:

```
uses gw.api.database.spatial.SpatialPoint
uses gw.api.database.Query

// Make a query on the Address entity.
var addressQuery = Query.make(Address)

// Define the location of the center point for the distance calculation.
var SAN_MATEO = new SpatialPoint(-122.300, 37.550)

// Specify the spatial point in the address instance and the maximum distance from the center.
addressQuery.withinDistance(Address.SPATIALPOINT_PROP.get(), SAN_MATEO, 10, UnitOfDistance.TC_MILE)
```

```
// Indicate that the query is an entity query by selecting all columns.  
var result = addressQuery.select()  
  
// Print the results of the query.  
for (address in result) {  
    print(address)  
}
```

The second code example uses the `withinDistance` method in connection with a more complex entity query that joins a primary entity type with a related entity type. This query joins the `Person` entity type, which has a `PrimaryAddress` property of type `Address`, with the `Address` entity type. The query then selects those `Person` entity instances with a `PrimaryAddress.SpatialPoint` property that is within 10 miles of a `SpatialPoint` constant representing San Mateo. The second code example is as follows:

```
uses gw.api.database.spatial.SpatialPoint  
uses gw.api.database.Query  
  
// Make a query on the Person entity.  
var personQuery = Query.make(Person)  
  
// Explicitly join the Address entity for the primary address to the Person.  
var addressTable = personQuery.join(Person#PrimaryAddress)  
  
// Define the location of the center point for the distance calculation.  
var SAN_MATEO = new SpatialPoint(-122.300, 37.550)  
  
// Specify the spatial point in the person's primary address and the maximum distance from the center.  
addressTable.withinDistance(Address.SPATIALPOINT_PROP.get(), "Person.PrimaryAddress.SpatialPoint", SAN_MATEO, 10,  
UnitOfDistance.TC_MILE)  
  
// Indicate that the query is an entity query by selecting all columns.  
var result = personQuery.select()  
  
// Print the results of the query.  
for (person in result) {  
    print(person + ", " + person.PrimaryAddress)  
}
```

Note: You cannot display the distance between the `SpatialPoint` property of an entity instance and a `SpatialPoint` constant in an entity query. To do so would require a call to the `Distance` static method in the `DBFunction` class, which is incompatible with entity queries. Only a row query can use a static method in the `DBFunction` class to return a value. To display distances in conducting a comparison, see “Using a distance function as a column selection” on page 783.

Combining predicates with AND and OR logic

Often you need more than one predicate in your query to select the items that you want in your result. For example, to select someone named “John Smith” generally requires you to specify two predicate expressions:

```
last_name = "Smith"  
first_name = "John"
```

To select the record that you need, both expressions must be true. You need a person’s last name to be “Smith” and that same person’s first name to be “John.” You do not want only one of the expressions to be true. You do not want to select people whose last name is “Smith” or first name is “John.”

For other requirements, you need to combine predicate expressions, any of which may be true, to select the items that you want. For example, to select addresses from Chicago or Los Angeles also generally requires two predicates:

```
city = "Chicago"  
city = "Los Angeles"
```

To select the addresses that you need, either expression may be true. You want an address to have either “Chicago” or “Los Angeles” as the city.

About Boolean AND and OR

Using AND to combine predicates that all must be true

In SQL, you combine predicates with the `AND` keyword if all must be true to include an item in the result.

```
WHERE last_name = "Smith" AND first_name = "John"
```

With the query builder APIs, you combine predicates by calling predicate methods on the query object one after the other if all must be true.

```
query.compare(Contact#LastName, Equals, "Smith")
query.compare(Contact#FirstName, Equals, "John")
```

You can also use the `and` method to make the `AND` combination of the predicates explicit. For example, the following code is equivalent to the previous example:

```
query.and( \ andCriteria -> {
    andCriteria.compare(Contact#LastName, Equals, "Smith")
    andCriteria.compare(Contact#FirstName, Equals, "John")
})
```

See also

- “Using Boolean algebra to combine sets of predicates” on page 764

Using OR to combine predicates that require at least one to be true

In SQL, you combine predicates with the `OR` keyword if one or more must be true to include an item in the result.

```
WHERE city = "Chicago" OR city = "Los Angeles"
```

With the query builder APIs, you combine predicates by calling the `or` method on the query object if one or more must be true.

```
query.or( \ orCriteria -> {
    orCriteria.compare(Address#City, Equals, "Chicago")
    orCriteria.compare(Address#City, Equals, "Los Angeles")
})
```

Using Boolean algebra to combine sets of predicates

You can use Boolean algebra to combine predicates with logical `AND` and `OR`. You use the `and` and `or` methods on queries, tables, and restrictions. The `and` and `or` methods modify the SQL query and add a clause to the query. To combine predicates, you use Gosu blocks, which are functions that you define in-line within another function.

Predicate linking mode

The default behavior of a new query is to use an implicit logical `AND` between its predicates. This behavior is known as a *predicate linking mode* of `AND`. The way that you apply `and` and `or` methods to a query can change the linking mode of a new series of predicates.

The `and` and `or` methods change only the linking mode of the predicates that are defined in the block that you pass to the method. The `and` and `or` methods do not change the linking mode of existing predicates. For example, if you apply multiple `or` methods directly to a query, the `OR` predicates are joined by an `AND`.

The `or` method has the following effect on the predicate linking mode of a query:

- To existing restrictions, add one parenthetical phrase.
- Link the new parenthetical phrase to previous restrictions in the current linking mode.
- The block passed to the `or` method includes a series of predicates. Use a predicate linking mode of `OR` between predicates inside the block.

- A predicates defined in the block could be another AND or OR grouping, containing additional predicates with another usage of the and or or methods.

The and method has the following effect on the predicate linking mode of a query:

- To existing restrictions, add one parenthetical phrase.
- Link the new parenthetical phrase to previous restrictions in the current linking mode.
- The block passed to the and method includes a series of predicates. Use a predicate linking mode of AND between predicates inside the block.
- A predicate defined in the block could be another AND or OR grouping, containing additional predicates with another usage of the and or or methods.

Syntax of Boolean algebra methods for predicates

The syntax of the or method is:

```
query.or( \ OR_GROUPING_VAR -> {
    OR_GROUPING_VAR.PredicateOrBooleanGrouping(...)
    OR_GROUPING_VAR.PredicateOrBooleanGrouping(...)
    [...]
})
```

The syntax of the and method is:

```
query.and( \ AND_GROUPING_VAR -> {
    AND_GROUPING_VAR.PredicateOrBooleanGrouping(...)
    AND_GROUPING_VAR.PredicateOrBooleanGrouping(...)
    [...]
})
```

Each use of *PredicateOrBooleanGrouping* could be either:

- A predicate method such as compare or between.
- Another Boolean grouping by calling the or or and method.

Within the block, you call the predicate and Boolean grouping methods on the argument that you pass to the block. Do not call the predicate and Boolean grouping methods on the original query.

In the syntax specifications, *OR_GROUPING_VAR* and *AND_GROUPING_VAR* refer to a grouping variable name that identifies the OR or AND link mode in each peer group of predicates. Guidewire recommends using a name for the block parameter variable that indicates one of the following:

Name describes the linking mode

Use a variable name that specifies the linking mode between predicates in that group, such as *or1* or *and1*. You cannot call the variable or or and because those words are language keywords. Add a digit or other unique identifier to the words or or and.

Name with specific semantic meaning

For example, use *carColors* for a section that checks for car colors.

Combining AND and OR groupings

You can combine and populate AND and OR groupings with any of the following:

- Predicate methods
- Database functions
- Subselect operations

If your query consists of a large number of predicates linked by OR clauses, the following alternative approaches might provide better performance in production, depending on your data:

- If the OR clauses all test a single property and all of the values are non-null, rewrite and collapse the tests into a single comparison predicate using the *compareIn* method, which takes a list of non-null values.

For example, if your query checks a property for a color value for matching any one of 30 color values, create a `Collection` that contains the color values. You can use any class that implements `Collection`, such as an array list or a set. Use the `Collection` as an argument to `compareIn`:

```
var lastNamesArrayList = {"Smith", "Applegate"}
var query = Query.make(Person)
query.compareIn(Person.LASTNAME_PROP.get(), lastNamesArrayList)

// Fetch the data with a for loop.
var result = query.select()
for (person in result) {
    print (person)
}
```

- Consider creating multiple subqueries and using a `union` clause to combine subqueries.

Consider trying your query with both approaches, and then test production performance with a large number of records.

Chaining inside AND and OR groupings

Typical predicate definitions use a pattern that uses one line of code for each predicate in the block. Using `or` in the name of the Gosu block variable more closely matches an English construction that describes the final output because an `OR` clause separates the predicates. Consider using the pattern in the following example, and avoid chaining the predicates together.

```
var query = Query.make(Person)

query.or( \ or1 -> {
    or1.compare(Person#CreateTime, GreaterThanOrEquals,
        DateUtil.addBusinessDays(DateUtil.currentDate(), -10))
    or1.compare(Person#LastName, Equals, "Newton")
    or1.compare(Person#FirstName, Equals, "Ray")
})
```

In theory, you could use method chaining to use a single line of code, rather than using three separate lines. In this case, chaining might make the code harder to understand.

```
var query = Query.make(Person)

query.or( \ or1 -> {
    or1.compare(Person#CreateTime, GreaterThanOrEquals, DateUtil.addBusinessDays(DateUtil.currentDate(),
        -10)).compare(Person#LastName, Equals, "Newton").compare(Person#FirstName, Equals, "Ray")
})
```

Examples

For example, the following Gosu code links three predicates together with logical OR. The query returns rows if any of the three predicates are true for that row.

```
var query = Query.make(Person)

// Find rows with creation date in the last 10 business days, last name of Newton, or first name of Ray
query.or( \ or1 -> {
    or1.compare(Person#CreateTime, GreaterThanOrEquals,
        DateUtil.addBusinessDays(DateUtil.currentDate(), -10))
    or1.compare(Person#LastName, Equals, "Newton")
    or1.compare(Person#FirstName, Equals, "Ray")
})
```

For example, the following Gosu code links three predicates together with logical AND. The query returns rows if all three predicates are true for that row.

```
var query = Query.make(Person)

// Find rows with last name of Newton, first name of Ray, and creation date more than 14 days ago
query.and( \ and1 -> {
    and1.compare(Person#LastName, Equals, "Newton")
    and1.compare(Person#FirstName, Equals, "Ray")
    and1.compare(Person#CreateTime, LessThanOrEquals, DateUtil.addDays(DateUtil.currentDate(), -14))
})
```

This code is functionally equivalent to simple linking of predicates, because the default linking mode is AND. The following Gosu code shows the use of predicate linking to achieve the same query.

```
var query = Query.make(Person)
query.compare(Person#LastName, Equals, "Newton")
query.compare(Person#FirstName, Equals, "Ray")
query.compare(Person#createTime, LessThanOrEquals, DateUtil.addDays(DateUtil.currentDate(), -14))
```

The power of the query building system is the ability to combine AND and OR groupings. For example, suppose we need to represent the following pseudo-code query predicates:

```
(CreateTime < 10 business days ago) OR ( (LastName = "Newton") AND (FirstName = "Ray") )
```

The following code represents this pseudo-code query by using Gosu query builders:

```
query.or( \ or1 -> {
    or1.compare(Person#createTime, LessThanOrEquals,
                DateUtil.addBusinessDays(DateUtil.currentDate(), -10))
    or1.and(\ and1 -> {
        and1.compare(Person#LastName, Equals, "Newton")
        and1.compare(Person#FirstName, Equals, "Ray")
    })
})
```

The outer OR contains two items, and the second item is an AND grouping. This structure directly matches the structure of the parentheses in the pseudo-code.

Similarly, suppose we need to represent the following pseudo-code query predicates:

```
( (WrittenDate < Today - 90 days) OR NOT ISNULL(CancellationDate) )
AND
( (BaseState = "FR") OR Locked )
```

The following code represents this pseudo-code using Gosu query builders:

```
query.or( \ or1 -> {
    or1.compare(DBFunction.DateFromTimestamp(or1.getColumnRef("WrittenDate")), LessThan,
                DateUtil.addDays(DateUtil.currentDate(), -90))
    or1.compare(PolicyPeriod#CancellationDate, NotEquals, Null)
})
query.or( \ or2 -> {
    or2.compare(PolicyPeriod#BaseState, Equals, Jurisdiction.TC_FR)
    or2.compare(PolicyPeriod#Locked, Equals, true)
})
```

Each OR contains two items, and the two OR items are combined using the default predicate linking mode of AND. This structure directly matches the structure of the parentheses in the pseudo-code.

See also

- *Gosu Reference Guide*

Joining a related entity to a query

To build useful queries in ClaimCenter, you often must join related entities to the primary entity of the query. Typical reasons to create a join are to restrict the set of rows that the query returns or to retrieve additional information that the related entities contain. The query builder APIs support setting restrictions only on database-backed fields, which store their values directly in the database. Primary entities have many properties that are arrays, foreign keys, type lists, and derived fields. Related tables provide the database-backed fields for these types of properties. When you join related entities to a query, you can set restrictions on the related fields, which in turn restricts the primary entities in the result.

Joining an entity to a query with a simple join

The term join comes from relational algebra. The term has special meaning for SQL and the query builder API.

Joining tables in SQL SELECT statements

In SQL, you can join two tables based on columns with matching values to form a new, virtual *join table*. The rows in the join table contain columns from each original table. On the join table, you can specify restrictions, which can have columns from both tables. When the SQL query runs, the result set has rows and columns from the join table. A join typically involves a column in one table that contains a foreign key value that matches a primary key value in a column in a second table. A *primary key* is a column with values that uniquely identify each row in a database table. A *foreign key* is a column in one table that contains values of a primary key in another table. The definition of a foreign key column specifies whether the values in the column are unique.

The following example SQL Select statement uses the `JOIN` keyword to join the `addresses` table to the `companies` table.

```
SELECT * FROM companies
JOIN addresses
ON addresses.ID = companies.primary_address;
```

In response to the preceding example SQL statement, the database returns a result set with all the rows in the `companies` table that have a primary address. The result set includes columns for the `companies` and columns for their primary addresses. The result set does not include companies without a primary address.

For example, the `companies` and `addresses` tables have the following rows of information:

`companies:`

<code>id</code>	<code>name</code>	<code>primary_address</code>
c:1	Hoffman Associates	a:1
c:2	Golden Arms Apartments	a:2
c:3	Industrial Wire and Chain	
c:4	North Creek Auto	a:3
c:5	Jamison & Sons	a:4

`addresses:`

<code>id</code>	<code>street_address</code>	<code>city</code>	<code>postal_code</code>
a:1	123 Main St.	White Bluff	AB-2450
a:2	45112 E. Maplewood	Columbus	EF-6370
a:3	3945 12th Ave.	Arlington	IB-4434
a:4	930 Capital Way	Arlington	IR-8775

The preceding example SQL statement returns the following result set.

`Result set with companies and addresses tables joined:`

<code>id</code>	<code>name</code>	<code>primary_address</code>	<code>id</code>	<code>street_address</code>	<code>city</code>	<code>postal_code</code>
c:1	Hoffman Associates	a:1		a:1 123 Main St.	White Bluff	AB-2450
c:2	Golden Arms Apartments	a:2		a:2 45112 E. Maplewood	Columbus	EF-6370
c:4	North Creek Auto	a:3		a:3 3945 12th Ave.	Arlington	IB-4434
c:5	Jamison & Sons	a:4		a:4 930 Capital Way	Arlington	IR-8775

The columns in the result set are a union of the columns in the `companies` and `addresses` tables. The company named “Industrial Wire and Chain” is not in the result set. That row in the `companies` table has `null` as the value for `primary_address`, so no row in the `addresses` table matches.

Joining entities with the query builder API

With the query builder API, you can join a related entity type to the primary entity type of the query. When you join an entity type to a query, the query builder API construct an internal object to represent the joined entity and its participation in the query.

In the simplest case, use the `join` method to join a dependent entity type to a query. You must specify the name of a property on the primary entity that the *Data Dictionary* defines as a foreign key to the dependent entity. Use the Gosu property name, not the actual column name in the database.

For example, the following Gosu code shows how to join the `Address` entity to a query of the `Company` entity. The code uses the `PrimaryAddress` property of `Company`, which is a foreign key to `Address`.

```
uses gw.api.database.Query

var query = Query.make(Company)
query.join(Company#PrimaryAddress) // Join Address entity to the query by a foreign key on Company
var select = query.select()

// Fetch the data with a for loop and print it.
for (company in select) {
    print(company)
}
```

Using the sample data shown earlier, the preceding query builder API query produces the following result.

Result object with Company instances from a join query:

Name

Hoffman Associates

Golden Arms Apartments

North Creek Auto

Jamison & Sons

The `Company` named “Industrial Wire and Chain” is not in the result object. This instance has `null` as its `primary_address`, so no instance of `Address` matches.

A significant difference from SQL is that the query builder API return only instances of the primary entity type. With the query builder API, you use dot notation to access information from the joined entity type. For example:

```
uses gw.api.database.Query

var query = Query.make(Company)
query.join(Company#PrimaryAddress) // Join Address entity to the query by a foreign key on Company
var select = query.select()

// Fetch the data with a for loop and print it --
for (company in select) {
    print(company + ", " + company.PrimaryAddress.City)
}
```

The preceding Gosu code fetches the `Address` instance lazily from the database. If the record is not available in the cache, this code could cause the execution of a query for every company. To reduce the number of database queries, you can use a row query to retrieve the complete set of rows and the necessary columns from the primary and joined tables.

See also

- “Working with row queries” on page 780

Ways to join a related entity to a query

In SQL, you can join a table to a query in these ways:

Inner join

Excludes rows/instances on the left that have no matches on the right.

Outer left join

Includes rows/instances from the left that have no matches on the right.

Outer right join

Include rows/instances from the right that have no matches on the left.

Full outer join

Include rows/instances that have no matches, regardless of side

The query builder APIs support inner joins and outer left joins only. Outer right joins and outer full joins are not supported.

Making a query with an inner join

With an inner join, items from the primary table or entity on the left are joined to items from the table or entity on the right, based on matching values. If a primary item on the left has no matching value on the right, that primary item is not included in the result. A foreign key from one side to the other is the basis of matching.

In SQL, it generally does not matter which side of the join provides the foreign key nor whether the foreign key is defined in metadata. All that matters is for one side of the join to have a column or property with values that match values in the column of another table or property.

The query builder API uses different signatures on the `join` method depending on which side of a join provides the foreign key.

- `join(primaryEntity#column)` – Joins two entities that have the foreign key is on left
- `join(secondaryEntity#column)` – Joins two entities that have the foreign key is on the right.
- `join("columnPrimary", table, "columnDependent")` – Joins two entities without regard to foreign keys.

Making an inner join with the foreign key on the left

In SQL, an inner join is the default type of join. You make an inner join with the `JOIN` keyword. To be explicit about the type of join, use `INNER JOIN`. You specify the table that you need to join to the primary table of the query. You use the `ON` keyword to specify the columns that join the tables. By convention, you specify the join column on the primary table first. In the following SQL statement, the foreign key, `PrimaryAddressID`, is on the primary table, `cc_contact`. Because SQL considers the primary table to be on the left side, the foreign key is on the left side of the join.

```
SELECT * FROM cc_contact
INNER JOIN cc_address
ON cc_address.ID = cc_contact.PrimaryAddressID;
```

With the query builder API, you use the `join` method to specify an inner join. Use a property reference of the primary entity and property name as the single parameter if the primary entity has the foreign key. In these cases, the foreign key is on the left. You specify a property of the primary entity that the *Data Dictionary* defines as a foreign key to the dependent entity. The query builder API uses metadata to determine the entity type to which the foreign key relates.

In the following Gosu code, the primary entity type, `Company`, has a foreign key, `PrimaryAddress`, which relates to the dependent entity type, `Address`.

```
var queryCompany = Query.make(Company)
queryCompany.join(Company#PrimaryAddress)
```

Unlike SQL, you do not specify which dependent entity to join, in this case `Address`. Neither do you specify the property on the dependent entity, `ID`. The query builder API uses metadata from the *Data Dictionary* to provide the missing information.

Making an inner join with the foreign key on the right

In SQL, an inner join is the default type of join. You make an inner join with the `JOIN` keyword. To be explicit about the type of join, use `INNER JOIN`. You specify the table that you need to join to the primary table of the query. You use

the ON keyword to specify the columns that join the tables. By convention, you specify the join column on the primary table first. In the following SQL statement, the foreign key, UpdateUserID, is on the secondary table, cc_address. Because SQL considers the primary table to be on the left side, the foreign key is on the right side of the join. The query returns a result with all addresses and users who last updated them. The result contains all columns in the address and user tables.

```
SELECT * FROM cc_user
INNER JOIN cc_address
ON cc_user.ID = cc_address.UpdateUserID;
```

With the query builder APIs, you use the `join` method to specify an inner join. Use a property reference of the dependent entity and property name as a single parameter if the dependent entity has the foreign key. In these cases, the foreign key is on the right. The database must have a column that contains the foreign key. The data model definition of the column must specify `foreignkey`. You cannot use a column that is a `onetoon` or `edgeForeignKey` type because these types do not use a database column. To confirm that a foreign key uses a database field, in the *Data Dictionary*, check that the foreign key does not have “(virtual property)” text after its name.

Joining to a dependent entity with the foreign key on the right

In the following Gosu code, the dependent entity type, `Address`, has a foreign key, `UpdateUser`, which relates to the primary entity type, `User`.

```
var queryUser = Query.make(User)
queryUser.join(Address#UpdateUser)
```

Similarly to SQL, you must specify the dependent entity to join, which in this case is `Address`. Unlike SQL, you do not specify the property on the primary entity, `ID`. The query builder APIs use metadata from the Data Dictionary to provide the missing information.

Applying predicates to the dependent entity

You seldom want to retrieve all instances of the primary type of a query. To select a subset, you often join a dependent entity and apply predicates to one or more of its properties. You apply these predicates for a secondary table with the foreign key on the right in the same way as for predicates where the foreign key is on the left.

For example, you want to select only users who have updated addresses in the city of Chicago. You must join the `Address` entity to your `User` query. Then, you can apply the predicate `City = "Chicago"` to `Address`, the dependent entity. The SQL for this query looks like the following lines.

```
SELECT DISTINCT cc_user.PublicID FROM cc_user
INNER JOIN cc_address
ON cc_user.ID = cc_address.UpdateUserID
AND cc_address.City = 'Chicago';
```

The following Gosu code represents the same SQL query.

```
uses gw.api.database.Query
uses gw.api.database.Relop

// -- Query the User entity --
var queryUser = Query.make(User)

// -- Select only User instances who last updated addresses in the city of Chicago --
var tableAddress = queryUser.join(Address#UpdateUser)
tableAddress.compare(Address#City, Equals, "Chicago")

// -- Fetch the User instances with a for loop and print them --
var result = queryUser.select()

for (user in result) {
    print (user.DisplayName)
}
```

See also

- “Restricting query results by using fields on joined entities” on page 776
- “Working with row queries” on page 780

- “Handling duplicates in joins with the foreign key on the right” on page 774

Making an inner join without regard to foreign keys

In SQL, you can make an inner join between two tables regardless of any formal SQL declaration of a foreign key relation between the tables. The columns on which a SQL Select statement joins two tables must have values that match, so the data types of both columns must be the same. In addition, application logic, not a declared foreign key constraint, must ensure that both columns contain values that potentially match.

To use the query builder API to create an inner join that does not use a foreign key, the database must have columns that contain each join property. The data model definition of the column must specify `foreignkey` or `column`. You cannot use a virtual column, nor a column that is an `array`, `onetoon`, or `edgeForeignKey` type because these types do not use a database column. To confirm that a foreign key uses a database field, in the *Data Dictionary*, check that the foreign key does not have “(virtual property)” text after its name.

Test the use of an inner join that you make without regard to foreign keys in a realistic environment. Using this technique on a query can cause a full-table scan in the database if the query cannot use an index.

WARNING: For a query on large tables, using an inner join without regard to foreign keys can cause an unacceptable delay to the user interface.

Comparing SQL syntax to query builder syntax

For an inner join in SQL, the `ON` clause specifies the columns that join the tables. The clause requires the names of both tables and both columns. The following SQL statement joins rows from the `addresses` table to rows in the `companies` table where values in `ID` match values in `primary_address`.

```
SELECT * FROM companies
JOIN addresses
  ON companies.primary_address = addresses.ID;
```

With the query builder API, you use the `join` method with the following signature to make inner joins of entities without regard to foreign keys in the data model.

```
join("columnPrimary", table, "columnDependent")
```

The first parameter is the name of a column on the primary entity of the query. The second and third parameters are the names of the dependent entity and the column that has matching values.

Example

In the following Gosu code, the primary entity `Note` has no declared foreign key relation with the dependent entity `Contact`. An indirect relation exists through their mutual foreign key relations with the `User` entity. Both entities in the query have a column that contains the user ID, so the `join` method can join the `Contact` entity to the query of the `Note` entity.

```
uses gw.api.database.Query

// -- query the Note entity --
var queryNote = Query.make(Note).withDistinct(true)

// -- select only notes where the same user created a contact.
queryNote.join("CreateUser", Contact, "CreateUser")

// -- fetch the Note instances with a for loop and print them --
var result = queryNote.select()

for (note in result) {
    print ("The user who created the note '" + note.Subject + "' also created a contact")
}
```

The `Note` entity has no direct relevant foreign key relation to `Contact`. You cannot use Gosu dot notation to navigate from notes in the result to properties on the contact.

Making a query with a left outer join

For a left outer join, items from the primary entity on the left are joined to items from the entity on the right, based on matching values. If a primary item on the left has no matching value on the right, that primary item is included in the result, anyway. A foreign key from one side or the other is the basis of matching.

In SQL, it generally does not matter which side of the join the foreign key is on. It generally does not matter whether the foreign key is defined in metadata. It does not matter if the column or property names are the same. All that matters is for one side of the join to have a column or property with values that match values in the column of another table or property.

Making a left outer join with the foreign key on the left

In SQL, you make a left outer join with the `LEFT OUTER JOIN` keywords. You specify the table that you want to join to the primary table of the query. The `ON` keyword lets you specify which columns join the tables. In the following SQL statement, the foreign key, `supervisor`, is on the primary table, `groups`. So, the foreign key is on the left side.

```
SELECT * FROM groups
LEFT OUTER JOIN users
ON groups.supervisor = users.ID;
```

With the query builder APIs, use the `outerJoin` method with a single parameter if the primary entity has the foreign key. In these cases, the foreign key is on the left. You specify a property of the primary entity that the Data Dictionary defines as a foreign key to the dependent entity. The query builder APIs use metadata to determine the entity type to which the foreign key relates.

Joining to a dependent entity with the foreign key on the left

In the following Gosu code, the primary entity type, `Group`, has a foreign key, `Supervisor`, which relates to the dependent entity type, `User`.

```
var queryGroup = Query.make(Group)
queryGroup.outerJoin(Group#Supervisor) // Supervisor is a foreign key from Group to User.
```

Notice that unlike SQL, you do not specify which dependent entity to join, in this case `User`. Neither do you specify the property on the dependent entity, `ID`. The query builder APIs use metadata from the Data Dictionary to fill in the missing information.

Accessing properties of the dependent entity

Unlike SQL, the result of an entity query contains instances of the primary entity, not composite table rows with columns. Explicitly specifying a join by using the `join` or `outerJoin` methods does not change how you access properties of the dependent entity. Information from joined entities generally is not included in results. If the foreign key is on the left, you can access properties from the dependent entity by using dot notation. However, if the foreign key is on the right, you cannot use dot notation to traverse from primary instances on the left to related instances on the right.

Dependent entity with the foreign key on the left

The following Gosu code has the foreign key on the left. The primary entity type, `Company`, has a foreign key, `PrimaryAddress`, which relates to the dependent entity type, `Address`. You can use dot notation to access the properties of `Address`.

```
uses gw.api.database.Query

var queryCompany = Query.make(Company)
queryCompany.join(Company#PrimaryAddress)

// Fetch the data with a for loop.
var result = queryCompany.select()
for (company in result) {
    print (company + ", " + company.PrimaryAddress.City)
}
```

Dependent entity with the foreign key on the right

The following Gosu code has the foreign key on the right. You can determine that a user updated addresses, but not which addresses were updated. You cannot use dot notation to retrieve information from the Address entity rows. This limitation does not affect processing the results of inner queries with foreign keys on the right if you need no information from the related instances that produced the result. If you do want to access the information, for example to display on a screen or in a report, you must use a row query.

```
uses gw.api.database.Query
// -- Query the User entity --
var queryUser = Query.make(User)

// -- Select only User instances who last updated addresses in the city of Chicago --
var tableAddress = queryUser.join(Address#UpdateUser)

// -- Fetch the User instances with a for loop and print them --
var result = queryUser.select()

for (user in result) {
    print (user.DisplayName)
}
```

Handling duplicates in joins with the foreign key on the right

Duplicate occurrences of records from the primary table in a result can occur only for joins that have the foreign key on the right. If the foreign key for a join is on the left, the value for any primary instance is unique in the dependent table. A foreign key on the left relates either to:

- A value of `null`, which matches no instance on the right
- A single instance on the right

A query that you create by using `join` or `outerJoin` with the foreign key on the right of the join often fetches duplicate instances of the primary entity. For example, the following Gosu code finds users that have updated addresses.

```
var queryParent = Query.make(User)
var tableChild = queryParent.join(Address#UpdateUser)
```

For the previous example, if multiple `Address` instances relate to the same `User`, the result includes duplicates of that `User` instance. One instance exists in the result of the join query for each child object of type `Address` that relates to that instance.

Duplicate instances of the primary entity in a query result are desirable in some cases. For example, if you intend to iterate across the result and extract properties from each child object that matches the query, this set of rows is what you want. In cases for which you need to return a single row from the primary table, you must design and test your query to ensure this result.

For joins with the foreign key on the right, try to reduce duplicates on the primary table. The best way to reduce duplicates is to ensure that the secondary table only has one row that matches the entity on the primary table.

If you cannot eliminate duplicates in this way, other approaches can limit duplicates created because of a join:

- Rewrite to use the `subselect` method approach. For joins with the key on the right, the query optimizer often performs better with `subselect` than with `join` or `outerJoin`. However, using the `join` method might perform better in some cases. For example, consider using `join` if the dependent entity includes predicates that are not very selective and return many results. For important queries, Guidewire recommends trying both approaches under performance testing. If you use `join` with the foreign key on the right, use the two-parameter method signature that includes the table name followed by the column name in the joined table.

```
var queryParent = Query.make(User)
queryParent.subselect(User#ID, InOperation.CompareIn, Note#Author)
```

- Call the `withDistinct` method on `Query` to limit the results to distinct results from the join. Pass the value `true` as an argument to limit the query results to contain only a single row for each row of the parent table. To turn off this behavior later, call the method again and pass `false` as an argument. For example:

```
var queryParent = Query.make(User)
queryParent.withDistinct(true)
queryParent.join(Note#Author)
```

- Add predicates to the secondary table to limit what your query matches. For example, only match entities on the primary table if a related child object has a certain property with a specific value or range of values.

The following example adds predicates after the join using properties on the joined table:

```
var queryParent = Query.make(User)
var tableChild = queryParent.join(Note#Author)
tableChild.compare(Note#createTime, GreaterThanOrEquals,
    DateUtil.addDays(DateUtil.currentDate(), -5))
```

See also

- “Working with row queries” on page 780

Joining to a subtype of an entity type

Many ClaimCenter entity types have subtypes. The parent entity type and its subtypes all share the same database table. For example, the `Contact` entity type, its direct subtypes, `Company`, `Person`, and `Place`, and its indirect subtypes all have rows in the `cc_contact` database table. If you create a query that accesses a subtype as the primary entity, the query builder API adds appropriate filters to the query to return only rows of that subtype. You use the same syntax to access properties on the parent type and properties that are specific to the subtype. If you create a query that joins a subtype as a dependent entity, the query builder API joins the least restrictive entity type to the query.

For example, the following lines join organizations to contacts:

```
var queryOrg = Query.make(Organization)
var tableContact = queryOrg.outerJoin(Organization#Contact)
```

Because the `Contact` property exists on the `Organization` entity type, the following code also joins organizations to contacts and does not join only company records:

```
var queryOrg = Query.make(Organization)
var tableContact = queryOrg.outerJoin(Company#Contact)
```

If you want to access company properties or only need to see organizations that have a company as a contact, you must cast the joined table to `Company`. Use code like the following lines:

```
var queryOrg = Query.make(Organization)
var tableContact = queryOrg.join(Organization#Contact).cast(Company)
```

If the column that you use to join the dependent table is specific to a particular subtype of an entity type, the query builder API performs the more restrictive join. In this case, you do not need to cast the type of the joined table.

Restricting a result set by joining to a subtype of an entity type

You can restrict the result set of a query by requiring a specific subtype for the joined entity. You use the entity name and a property to specify the join. The query builder API generates an SQL query to join to the database table. If the property for the join exists on the subtype entity and not on the more general entity type, the generated SQL query adds a restriction for the subtype. If the property for the join exists on the more general entity type, the generated SQL query does not add a restriction for the subtype. To restrict the rows in the result set to those matching the subtype of the joined table, you must use the `cast` method.

For example, the following code prints the display names of claims for one or more injury incidents:

```
uses gw.api.database.Query
var queryClaim = Query.make(Claim).withDistinct(true)
```

```
// Join to any type of Incident
// queryClaim.join(Incident#Claim)           // This join does not select only injury incidents

// Join to only injury incidents
queryClaim.join(Incident#Claim).cast(Incident) // This join does select only injury incidents
var result = queryClaim.select()

for (claim in result) {
    print(claim)
}
```

Accessing properties of a joined subtype of an entity type

If you need to access the properties of a specific subtype of an entity type, you must ensure that the query returns only rows for that subtype. If the subtype is the primary entity, you make a query by specifying the subtype name. If the subtype is a joined entity and the join uses a property on the parent entity type, you must cast the table to the required subtype. Additionally, if you need to access a property on the joined entity subtype, you must cast the property on the primary entity to the specific subtype.

For example, the following code prints information about activities and the person who is the contact for a claim:

```
uses gw.api.database.Query

var queryActivity = Query.make(Activity)

// Join only to Person contacts
var tablePerson = queryActivity.join(Activity#ClaimContact).join(ClaimContact#Contact).cast(Person)

var result = queryActivity.select()

for (activity in result) {
    // Use a variable to access properties on the Person entity type
    var co = activity.ClaimContact.Contact as Person
    print(activity.UpdateUser + " " + activity + " " + co + " " + co.CellPhone)
}
```

Restricting query results by using fields on joined entities

You seldom want to retrieve all instances of the primary type of a query. In many cases, you join a dependent entity to a query so you can restrict the results by applying predicates to one or more of its properties. To apply predicates to the dependent entity in a join, you can save the `Table` object that the `join` method returns in a local variable. Then, you can use predicate methods on the `Table` object, such as `compare`, to restrict which dependent instances to include in the result. That restriction also restricts the primary entity instances that the result returns.

Applying predicates to the dependent entity of an inner join

For example, you want a query that returns all companies in the city of Chicago. The SQL statement for this query looks like:

```
SELECT * FROM cc_contact
JOIN cc_address
    ON cc_address.ID = cc_contact.PrimaryAddressID
WHERE cc_contact.Subtype IN (2,6,7,10,12,14)
    AND cc_address.City = 'Chicago';
```

Using the query API, the primary entity type of your query is `Company`, because you want instances of that type in your result.

```
var queryCompany = Query.make(Company)
```

The `Company` entity does not have a `City` property. That property is on the `Address` entity. You need to join the `Address` entity to your query. You must capture the object reference that the `join` method returns so that you can specify predicates on values in `Address`.

```
var tableAddress = queryCompany.join(Company#PrimaryAddress)
tableAddress.compare(Address#City, Equals, "Chicago")
```

Alternatively, you can chain the `join` method and call the `predicate` method on the returned `Table` object in a single statement, as the following Gosu code shows. If you use this pattern, you do not need to make a variable for the `Table` object.

```
queryCompany.join(Company#PrimaryAddress).compare(Address#City, Equals, "Chicago")
```

When you run the query, the result contains companies that have Chicago as the city on their primary addresses. The following code demonstrates the use of this predicate on a joined entity.

```
uses gw.api.database.Query
uses gw.api.database.Relop

// Start a new query with Company as the primary entity.
var queryCompany = Query.make(Company)

// Join Address as the dependent entity to the primary entity Company.
var tableAddress = queryCompany.join(Company#PrimaryAddress)

// Add a predicate on the dependent entity.
tableAddress.compare(Address#City, Equals, "Chicago")

// Fetch the data with a for loop.
var result = queryCompany.select()
for (company in result) {
    print (company + ", " + company.PrimaryAddress.City)
}
```

Running this code produces a list like the following one:

```
Jones and West Insurance, Chicago
Shumway Contracting, Chicago
New Energy Corp., Chicago
...
Auto Claims of the West, Chicago
Auto Claims Defenders, Chicago
```

The query result of the previous example contains only `Company` instances. Even though the code includes the `Address` entity in the query, the results do not include information from joined entities. The code example uses dot notation to access information from related `Address` instances after retrieving `Company` instances from the result. By joining `Address` to the query and applying the predicate, the code ensures that `Company` instances in the result have only “Chicago” in the property `company.PrimaryAddress.City`.

Predicates on the left outer join dependent entity differ from SQL

With the query builder APIs, generally you join a dependent entity to a query only so you can restrict the results with predicates on the dependent entity. Using the `compare` method to apply a predicate to a dependent entity that you join with a left outer join has no effect on the rows that Gosu retrieves.

The ways that SQL queries and Gosu code filter the dependent table rows differ. Gosu fetches the dependent entity only if necessary, if code that accesses the dependent entity executes. Gosu applies the filter when running the query on the primary table, not when fetching the dependent table row. The value of a dependent entity property that you access by using dot notation is `null` only if there is no matching entity instance regardless of the filter. To retrieve filtered values from related entities, use a row query.

For example, the following SQL filters the names of group supervisors. If the name does not match, the value for the supervisor user and contact columns are `null`.

```
SELECT * FROM groups
LEFT OUTER JOIN users
    ON groups.supervisor = users.ID
LEFT OUTER JOIN contacts
    ON users.contactID = contacts.ID
    AND contacts.lastName = 'Visor';
```

The following Gosu code shows the effect of dot notation to access information related to the primary entity of an outer-join query. The code accesses the `DisplayName` information for the group supervisor from the `User` entity with dot notation on `Group` instances retrieved from the result. The output shows the supervisor name for every group

because Gosu applies the filter on the primary table query, not when print retrieves the property values for related entities.

```
uses gw.api.database.Query

// -- Query the Group entity --
var queryGroup = Query.make(Group)
// -- Supervisor is a foreign key from Group to User.
var userTable = queryGroup.outerJoin(Group#Supervisor)
// -- Contact is a foreign key from User to UserContact.
var contactTable = userTable.outerJoin(User#Contact)
// -- Apply a filter to the outer joined entity
contactTable.compare(UserContact#LastName, Relop.Equals, "Visor")
// -- Fetch the Group instances with a for loop and print them --
var result = queryGroup.select()

for (group in result) {
    if (group.Supervisor != null) {
        // Every supervisor's name appears because the outer join does not filter the groups
        print(group.Name + ": " + group.Supervisor.Contact.DisplayName)
    } else { // Provide user-friendly text if the group has no supervisor
        // This text appears only for groups that have no supervisor
        print (group.Name + ": " + "This group has no supervisor")
    }
}
```

See also

- “Restricting a query with predicates on fields” on page 751
- “Restricting query results with fields on primary and joined entities” on page 778
- “Working with row queries” on page 780

Restricting query results with fields on primary and joined entities

You can restrict the result set of a query by applying predicates to both the primary and joined entities. You use different techniques to add the predicates depending on whether you want both (AND) or either (OR) to apply.

Using AND to combine predicates on primary and joined entities

If you need both predicates to apply to the result set, you can use a predicate method on the primary query and another predicate method on the secondary table.

For example, you want a query that returns all companies having a name that matches “Stewart Media” in the city of Chicago. The SQL statement for this query looks like:

```
SELECT * FROM cc_contact
JOIN cc_address
    ON cc_address.ID = cc_contact.PrimaryAddressID
WHERE cc_contact.Subtype IN (2,6,7,10,12,14)
    AND cc_contact.Name = 'Stewart Media'
    AND cc_address.City = 'Chicago';
```

Using the query API, the primary entity type of your query is **Company**, because you want instances of that type in your result. You add the predicate that specifies the restriction on the **Name** property.

```
var queryCompany = Query.make(Company).compare(Company#Name, Equals, "Stewart Media")
```

The **Company** entity does not have a **City** property. That property is on the **Address** entity. You need to join the **Address** entity to your query. You must capture the object reference that the **join** method returns so that you can specify predicates on values in **Address**.

```
var tableAddress = queryCompany.join(Company#PrimaryAddress)
tableAddress.compare(Address#City, Equals, "Chicago")
```

When you run the query, the result contains companies that have Chicago as the city on their primary addresses.

```
uses gw.api.database.Query
uses gw.api.database.Relop

// Start a new query with Company as the primary entity and restrict the result by company name.
```

```

var queryCompany = Query.make(Company).compare(Company#Name, Equals, "Stewart Media")
// Join Address as the dependent entity to the primary entity Company.
var tableAddress = queryCompany.join(Company#PrimaryAddress)

// Add a predicate on the dependent entity.
tableAddress.compare(Address#City, Equals, "Chicago")

// Fetch the data with a for loop.
var result = queryCompany.select()
for (company in result) {
    print(company + ", " + company.PrimaryAddress.City)
}

```

Running this code produces a list like the following one:

```
Stewart Media, Chicago
```

You can add predicates to the primary entity of a query before or after you join a new entity to the query. For example, the following code adds a predicate to the primary entity before joining the secondary entity:

```

uses gw.api.database.Query
uses gw.api.database.Relop

// -- Query the Contact entity --
var queryContact = Query.make(Contact)
// -- Select only contacts that have a primary phone number that is a work phone
queryContact.compare(Contact#PrimaryPhone, Equals, typekey.PrimaryPhoneType.TC_WORK)
// -- Select only Contact instances that have a primary address in the city of Chicago --
queryContact.join(Contact#PrimaryAddress).compare(Address#City, Equals, "Chicago")

```

The following code adds a predicate to the primary entity after joining the secondary entity, and is equivalent to the previous code example:

```

uses gw.api.database.Query
uses gw.api.database.Relop

// -- Query the Contact entity --
var queryContact = Query.make(Contact)

// -- Add a predicate on the secondary entity
// -- Select only Contact instances that have a primary address in the city of Chicago --
var tableAddress = queryContact.join(Contact#PrimaryAddress)
tableAddress.compare(Address#City, Equals, "Chicago")

// -- Add a predicate on the primary entity
// -- Select only contacts that have a primary phone number that is a work phone
queryContact.compare(Contact#PrimaryPhone, Equals, typekey.PrimaryPhoneType.TC_WORK)

```

Using OR to combine predicates on primary and joined entities

If you need either predicate to apply to the result set, you use an `or` method on the secondary table. To refer to a column on the secondary table in the `or` block, you can use a property reference. To refer to a column on the primary table in the `or` block, you must use a column reference. A property on the primary table is not available in the context of the secondary table.

For example, you want a query that returns all companies either having a name that matches “Armstrong Cleaners” or in the city of Chicago. The SQL statement for this query looks like:

```

SELECT * FROM cc_contact
JOIN cc_address
ON cc_address.ID = cc_contact.PrimaryAddressID
WHERE cc_contact.Subtype IN (2,6,7,10,12,14)
AND (cc_contact.Name = 'Armstrong Cleaners'
OR cc_address.City = 'Chicago');

```

Using the query API, the primary entity type of your query is `Company`, because you want instances of that type in your result.

```
var queryCompany = Query.make(Company)
```

The `Company` entity does not have a `City` property. That property is on the `Address` entity. You need to join the `Address` entity to your query. You must capture the object reference that the `join` method returns so that you can specify predicates on values in `Address`. You use an `or` block to add the predicate that specifies the restriction on the

Name property in the Company entity. You must use a column reference to access the Name column because that column is not in the Address entity.

```
var tableAddress = queryCompany.join(Company#PrimaryAddress)
tableAddress.compare(Address#City, Equals, "Chicago")
```

When you run the query, the result contains companies that have Chicago as the city on their primary addresses.

```
uses gw.api.database.Query
uses gw.api.database.Relop

// Start a new query with Company as the primary entity.
var queryCompany = Query.make(Company)

// Join Address as the dependent entity to the primary entity Company.
var tableAddress = queryCompany.join(Company#PrimaryAddress)

// Add both predicates on the dependent entity.
tableAddress.or( \ or1 -> {
    or1.compare(Address#City, Equals, "Chicago")
    or1.compare(queryCompany.getColumnRef("Name"), Equals, "Armstrong Cleaners")
})

// Fetch the data with a for loop.
var result = queryCompany.select()
for (company in result) {
    print (company + ", " + company.PrimaryAddress.City)
}
```

Running this code produces a list like the following one:

```
Armstrong Cleaners, San Ramon
Jones and West Insurance, Chicago
Shumway Contracting, Chicago
New Energy Corp., Chicago
...
Auto Claims of the West, Chicago
Auto Claims Defenders, Chicago
```

See also

- “Restricting a query with predicates on fields” on page 751
- “Restricting query results by using fields on joined entities” on page 776

Working with row queries

Certain kinds of SQL results require row queries instead of entity queries. The results of row queries are row objects that provide only the properties that you select.

Entity queries cannot produce certain kinds of SQL results. Entity queries have the following characteristics:

- They cannot produce the results that SQL aggregate functions provide.
- They cannot produce results that involve filtering the rows in the dependent table on a SQL outer join.
- They load the complete object row into the application cache for selected entity instances, even though you often need only a few columns of data from the database.
- They provide access to related entity instances by making additional queries to the database to access entity instances that are not in the cache. These queries also load entire rows from the database tables.

In many cases, a row query can provide more suitable results than an entity query. A row query uses a single database query on both the main entity and related entities to provide only the subset of properties that you need.

If you need to view the results of your query in a list view, you must convert a row query to another format. List views cannot interpret the results of a row query. Consider whether an entity query on a view entity can provide more straightforward access to a subset of columns on primary and related entities.

See also

- “Limitations of row queries” on page 787

- Configuration Guide

Setting up row queries

You set up row queries by passing column selection parameters to the `select` method on query objects. The columns that you select become the columns of data in the row query result. The columns that you select for a row query need not be the columns to which you applied predicates prior to calling the `select` method. The columns that you select are the columns of data that you want to access from rows in the row query result.

Names for selected columns

You can always access values for selected columns in a row query result by their position in the list of columns. Alternatively, you can access values by using aliases that you provide when you select the columns. An *alias* is a `String` value of your choice, which serves as a key for accessing specific column values from individual rows in the result. If you do not specify an alias, the default name of a column has the `entity.property` syntax. An alias replaces this default name.

Aggregate functions in row queries

You set up an *aggregate row query* by applying database aggregate functions to selected columns. For an aggregate row query, the result contains values that are aggregated from the table rows used in the query instead of the rows themselves. The database performs the aggregation as part of executing the query.

Selecting columns for row queries

Select the columns for row queries by passing a variable length list of column selection arguments to the `select` method on query objects. Whenever you pass column selection arguments to the `select` method, you set up a row query. Only the `select` method on query objects can set up a row query. The `select` methods on table and restriction objects accept only empty argument lists. An empty argument list passed to the `select` method sets up an entity query instead of a row query. The syntax for selecting columns for a row query is:

```
Query.select({Query Select Column[, Query Select Column...]})
```

See also

- “Applying a database function to a column” on page 782
- “Paths” on page 808
- “Aggregate functions in the query builder APIs” on page 816

Creating column specifications

The column selection arguments that you pass to the `select` method for a row query are objects that implement `IQuerySelectColumn`. You use methods on the `QuerySelectColumns` class to create column selection arguments. The columns that you select must be columns in the database. You cannot specify virtual properties, enhancement methods, or other entity methods as column selections. To create a column that you reference by `entity.property` syntax or by position, starting from 0, you use the `path` method on `QuerySelectColumns`. To create a named column, you use the `pathWithAlias` method.

You use methods on the `gw.api.path.Paths` class to create column paths, which are `gw.api.path.PersistentPath` objects. To specify the path to an individual column, you use the `make` method on `Paths`. To specify the path to a column in a related entity, you provide additional parameters to the `make` method. For example, the first two of the following Gosu expressions select the `ID` column from the `Contact` table. The second expression sets the alias `ContactID` on the column. The third expression creates a column for the user name of the user that created the `Contact` entity instance:

```
QuerySelectColumns.path(Paths.make(Contact#ID))
QuerySelectColumns.pathWithAlias("ContactID", Paths.make(Contact#ID))
QuerySelectColumns.path(Paths.make(Contact#createUser, User#Credential, Credential#userName))
```

The following Gosu code sets up a row query that returns the columns Subtype, FirstName, and LastName for all Person entity instances and prints the values for each row.

```
uses gw.api.database.Query
uses gw.api.database.QuerySelectColumns
uses gw.api.path.Paths

var query = Query.make(Person)

var results = query.select({
    QuerySelectColumns.path(Paths.make(Person#Subtype)),
    QuerySelectColumns.pathWithAlias("FName", Paths.make(Person#FirstName)),
    QuerySelectColumns.pathWithAlias("LName", Paths.make(Person#LastName))
})

for (row in results) {
    // Access the first column by entity.property, the second column by alias, the third by position
    print(row.getColumn("Person.Subtype") + ":" + row.getColumn("FName") + " " + row.getColumn(2))
}
```

Using foreign key properties and type key properties in row queries

Selecting a foreign key or type key property as a column in a row query does not provide access to the properties of the entity instance or type code. If you select a column that is a foreign key to a record in another database table, the column provides the identifier value of that record. The type of the foreign key column is `gw.pl.persistence.core.Key`. If you select a column that is a type key, the column provides the value of the type code. The type of the type key column is the corresponding typekey class. For example, the type of an address type typekey is `typekey.AddressType`.

The following Gosu code sets up a row query that returns the columns AddressType and CreateUser for all Address entity instances and prints the values for each row.

```
uses gw.api.database.Query
uses gw.api.database.QuerySelectColumns
uses gw.api.path.Paths

var query = Query.make(Address)

var results = query.select({
    // The Type column has a type of typekey.AddressType
    QuerySelectColumns.pathWithAlias("Type", Paths.make(Address#AddressType)),
    // The User column has a type of gw.pl.persistence.core.Key
    QuerySelectColumns.pathWithAlias("CUser", Paths.make(Address#CreateUser))
})

for (row in results) {
    // Access the type code value and the entity instance ID
    print(row.getColumn("Type") + " " + row.getColumn("CUser"))
}
```

The output from this code looks like the following lines.

```
business null
business 3
...
home 9
...
```

See also

- “Transforming results of row queries to other types” on page 785
- “Accessing data from virtual properties, arrays, or keys” on page 786

Applying a database function to a column

You often create row queries because you want to apply a database function, such as Sum or Max aggregates, to the values in a column for selected rows. The following SQL statement is a query that uses a maximum aggregate:

```
SELECT Country, MAX(CreateTime)
  FROM cc_Address
 GROUP BY Country;
```

To create a positional column that you reference by number, starting from 0, you use the `path` method on `QuerySelectColumns`. To create a named column, you use the `pathWithAlias` method. You use methods on the `DBFunction` class to apply a database aggregate function to a column. For example, the following Gosu expressions both select the maximum value in the `CreateTime` column from the `Address` table. The second expression gives the alias `NewestAddr` to the column.

```
QuerySelectColumns.dbFunction(DBFunction.Max(Paths.make(Address#CreateTime)))
QuerySelectColumns.dbFunctionWithAlias("NewestAddr", DBFunction.Max(Paths.make(Address#CreateTime)))
```

The database performs the aggregation as part of executing the query.

The query builder API inserts a `GROUP BY` clause in the generated SQL if necessary. The arguments to the `GROUP BY` clause are the non-aggregate columns that you provide to the `select` method. If you specify no non-aggregate columns, the query builder does not add a `GROUP BY` clause.

The database aggregate functions on the `DBFunction` class are `Sum`, `Max`, `Min`, `Avg`, and `Count`. The `DBFunction` class does not provide `First` or `Last` functions. The equivalent of the `First` function is the `FirstResult` property of the results that the `select` method returns. You can use the `FirstResult` property to simulate the `Last` function by applying the `orderByDescending` method on the results. Using `FirstResult` provides better query performance than stopping a query iteration after the first row.

Using an aggregate function as a column selection

The following Gosu code uses the `Max` database aggregate function. The query builder API appends a `GROUP BY Country` clause to the SQL query that the database receives.

```
uses gw.api.database.Query
uses gw.api.database.QuerySelectColumns
uses gw.api.path.Paths
uses gw.api.database.DBFunction

var query = Query.make(Address)
var latestAddress = QuerySelectColumns.dbFunctionWithAlias("LatestAddress",
    DBFunction.Max(Paths.make(Address#CreateTime)))
var rowResults = query.select({
    QuerySelectColumns.pathWithAlias("Country", Paths.make(Address#Country)),
    latestAddress
})
```

The query returns the time of the most recent address creation for each country.

See also

- “Aggregate functions in the query builder APIs” on page 816

Using a distance function as a column selection

The following two Gosu code examples use the `Distance` database function.

The first code example constructs a query that returns all unique addresses within ten miles of San Mateo along with their respective distances from San Mateo in miles. The code example then prints the results of the query.

In particular, the code example sets up a row query, `addressQuery`. For all distinct `Address` entity instances in the database, this row query returns the following columns: `Line1`, `City`, `State`, `PostalCode`, and `Distance`. Constructing the last column, `Distance`, involves calling the `Distance` method.

The `Distance` method returns a simple function. This function returns the distance between the location of an entity instance and a common reference location.

As a first parameter, the `Distance` method takes a row query with a proximity or spatial restriction. Due to a call to the `withinDistance` method, the `addressQuery` row query qualifies as having such a restriction.

As a second parameter, the `Distance` method takes the `String` name of a location property. The `addressQuery` row query rows represent `Address` entity instances. These entity instances all have a `SpatialPoint` location property. Hence, “`Address.SpatialPoint`” is the `String` name of a location property.

The `Distance` method uses these two parameters to calculate the distance between each location that the `SpatialPoint` property values represent and the reference location for the restriction, San Mateo. The resulting distances supply the `Distance` column with values. The first code example is as follows:

```

uses gw.api.database.DBFunction
uses gw.api.path.Paths
uses gw.api.database.spatial.SpatialPoint
uses gw.api.database.Query
uses gw.api.database.QuerySelectColumns

// Make a query on the Address entity. Remove duplicate entities from the query.
var addressQuery = Query.make(Address).withDistinct(true)

// Define the location of the center point for the distance calculation.
var SAN_MATEO = new SpatialPoint(-122.300, 37.550)

// Specify the spatial point in the address instance and the maximum distance from the center.
addressQuery.withinDistance(Address.SPATIALPOINT_PROP.get(), SAN_MATEO, 10, UnitOfDistance.TC_MILE)

// Define the columns for the row query.
var result = addressQuery.select({
    QuerySelectColumns.pathWithAlias("Line1", Paths.make(Address#AddressLine1)),
    QuerySelectColumns.pathWithAlias("City", Paths.make(Address#City)),
    QuerySelectColumns.pathWithAlias("State", Paths.make(Address#State)),
    QuerySelectColumns.pathWithAlias("PostalCode", Paths.make(Address#PostalCode)),
    QuerySelectColumns.dbFunctionWithAlias("Distance", DBFunction.Distance(addressQuery, "Address.SpatialPoint"))
})

// Print the results of the query.
for (address in result) {
    print(address.getColumn("Line1") + ", "
        + address.getColumn("City") + ", "
        + address.getColumn("State") + ", "
        + address.getColumn("PostalCode")
        + " is " + address.getColumn("Distance") + " miles away from San Mateo")
}

```

The second code example constructs a row query, `personQuery`. This query returns the name, primary address, and distance from San Mateo for all people in the database having a primary address that is within ten miles of San Mateo. The code example then prints the results of the query.

The call to the `Distance` method takes the row query, `personQuery`, as a first parameter. As a second parameter, the `Distance` method takes the String name of the location property that pinpoints the primary address of each identified Person. This String name is `"Person.PrimaryAddress.SpatialPoint"`. The second code example is as follows:

```

uses gw.api.database.DBFunction
uses gw.api.path.Paths
uses gw.api.database.spatial.SpatialPoint
uses gw.api.database.Query
uses gw.api.database.QuerySelectColumns

// Make a query on the Person entity.
var personQuery = Query.make(Person)

// Explicitly join the Address entity for the primary address to the Person.
var addressTable = personQuery.join(Person#PrimaryAddress)

// Define the location of the center point for the distance calculation.
var SAN_MATEO = new SpatialPoint(-122.300, 37.550)

// Specify the spatial point in the person's primary address and the maximum distance from the center.
addressTable.withinDistance(Address.SPATIALPOINT_PROP.get(), "Person.PrimaryAddress.SpatialPoint", SAN_MATEO, 10,
    UnitOfDistance.TC_MILE)

// Define the columns for the row query.
var result = personQuery.select({
    QuerySelectColumns.path(Paths.make(Person#FirstName)),
    QuerySelectColumns.path(Paths.make(Person#LastName)),
    QuerySelectColumns.pathWithAlias("Line1", Paths.make(Person#PrimaryAddress, Address#AddressLine1)),
    QuerySelectColumns.pathWithAlias("City", Paths.make(Person#PrimaryAddress, Address#City)),
    QuerySelectColumns.pathWithAlias("State", Paths.make(Person#PrimaryAddress, Address#State)),
    QuerySelectColumns.pathWithAlias("PostalCode", Paths.make(Person#PrimaryAddress, Address#PostalCode)),
    QuerySelectColumns.dbFunctionWithAlias("Distance", DBFunction.Distance(personQuery,
        "Person.PrimaryAddress.SpatialPoint"))
})

// Print the results of the query.
for (address in result) {
    print(
        address.getColumn("Person.FirstName") + " "
        + address.getColumn("Person.LastName") + ", "
        + address.getColumn("Line1") + ", "
        + address.getColumn("City") + ", "
    )
}

```

```
+ address.getColumn("State") + ", "
+ address.getColumn("PostalCode")
+ " is " + address.getColumn("Distance") + " miles away from San Mateo")
}
```

Creating and using an SQL database function

The `Expr` method on the `DBFunction` class returns a function defined by a list of column references and character sequences. The argument to this function is a list that contains only objects of type:

- `java.lang(CharSequence` – For example, pass a `String` that contains SQL operators or other functions.
- `gw.api.database.ColumnRef` – The type that the `query.getColumnRef(columnName)` method returns.

The query builder APIs concatenate the objects in the list in the order specified to form an SQL expression.

For example, the following Gosu code creates a new database function from column references to two columns, with the sum (+) operator. You can use the new function to compare values against the sum of these two columns.

```
// Create an SQL function that subtracts two integer properties
var query = Query.make(Person).withLogSQL(true)
var expr = DBFunction.Expr({
    query.getColumnRef("NumDependents") , " - ", query.getColumnRef("NumDependentsU18")
})
query.compare(Person#NumDependents, Relop.NotEquals, null)
query.compare(Person#NumDependentsU18, Relop.NotEquals, null)
query.compare(expr, GreaterThan, DBFunction.Constant(2))

var results = query.select()
results.orderBy(QuerySelectColumns.path(Paths.make(Person#FirstName)))
    .thenBy(QuerySelectColumns.path(Paths.make(Person#LastName)))

print("Rows where Person has more than 2 dependents aged 18 or over")
for (row in results) {
    print(row.DisplayName + " Total Dependents " + row.NumDependents +
        ", Dependents 18 or over " + (row.NumDependents - row.NumDependentsU18))
}
```

This code prints results similar to the following:

```
Rows where Person has more than 2 dependents aged 18 or over
Adam Auditor Total Dependents 5, Dependents 18 or over 3
Alex Griffiths Total Dependents 4, Dependents 18 or over 3
Alice Shiu Total Dependents 5, Dependents 18 or over 5
Allison Whiting Total Dependents 6, Dependents 18 or over 5
Annie Turner Total Dependents 5, Dependents 18 or over 3
...
```

Transforming results of row queries to other types

The default type of a row query result is an instance of a class that implements `IQueryResult`. This object provides zero or more row objects of type `QueryRow<Object>`. The `IQueryResult` object is iterable. You convert this result to another type for special purposes, such as using a row query result as the data source for a list view in a page configuration file.

You can specify whatever return type is most convenient for your program. The type does not need to match the native types for the columns in the database.

IMPORTANT: Choose the type that you want for the result carefully. Performance characteristics vary depending on the type that you choose.

Transforming row query results to Gosu or Java objects

You can use the `transformQueryRow` method on the result object of a row query to obtain a query result of Gosu or Java objects. For example, you can transform multiple columns to a Gosu data transfer object or a single column to a Java `String`. The return value of `transformQueryRow` is an instance of a class that implements `IQueryResult`. This

object provides zero or more objects of the transformed type. For example, the following Gosu code returns an `IQueryResult` object that contains objects of type `ShortContact`.

```

uses gw.api.database.Query
uses gw.api.database.QuerySelectColumns
uses gw.api.path.Paths

var query = Query.make(Person).withLogSQL(true)

var results = query.select({
    QuerySelectColumns.pathWithAlias("Subtype", Paths.make(Person#Subtype)),
    QuerySelectColumns.pathWithAlias("PName", Paths.make(Person#FirstName))
}).transformQueryRow(\row -> {
    // Create a ShortContact object for each row
    return new ShortContact(
        row.getColumn("Subtype") as typekey.Contact, row.getColumn("PName") as String)
})

for (p in results index i) {
    print ((i +1) + ":" + p.Subtype.Description + " " + p.Name)
}

class ShortContact {
    var _subType : typekey.Contact as Subtype
    var _name : String as Name
    construct(st : typekey.Contact, n : String){
        _subType = st
        _name = n
    }
}

```

Sometimes you only need a single column from each entity. In these cases, you use a simpler syntax for the argument to `transformQueryRow`. For example, to change the preceding code to retrieve just the `PName` column instead of a `ShortContact` object, use the following line that returns an `IQueryResult<Person, String>` object.

```
transformQueryRow(\row -> row.getColumn("PName") as String)
```

Transforming row query results to collections and lists

The result of a row query returned by the `select` method with parameters is an `IQueryResult`. A row query result is iterable, so you can use it directly in a `for` loop without transformation.

If you need a different iterable type, you can convert the query result objects of row queries to lists, arrays, collections, and sets by using the conversion methods:

- `toCollection`
- `toList`
- `toSet`
- `toTypedArray`

For example, you can use a conversion method on the returned value from the `transformQueryRow` method to produce a `Collection` of data transfer objects.

These conversion methods run the query and materialize the results into memory. If your result set is large, running these methods can exceed the available memory. For large result sets, best practice is to set the page size to prefetch query results.

The `where` method is not applicable on lists, arrays, collections, or sets returned by conversion methods on result objects. Instead, apply all selection criteria to query objects by using predicate methods before calling the `select` method.

See also

- *Gosu Reference Guide*

Accessing data from virtual properties, arrays, or keys

A row query provides access to specific properties of primary and secondary entities that are backed by columns in the database.

A row query cannot directly access virtual properties. The values of virtual properties are provided by methods rather than directly from the database.

A row query can access secondary entities from foreign keys or edge foreign keys. A row query cannot directly access arbitrary properties of secondary entities from edge foreign keys or foreign keys. A row query can provide only the identifier value that these properties contain or specified properties of the secondary entity.

A row query cannot access array properties or one-to-one properties. A row query cannot directly access any properties of secondary entities from array properties or one-to-one properties.

The most straightforward way to access these types of values is to use an entity query instead of a row query.

Alternatively, you can use a bundle. This approach performs a query on the database for each row in the result set to retrieve the necessary entities. The following code demonstrates the use of a bundle and keys to access a virtual property and entities in an array.

```
uses gw.api.database.Query
uses gw.api.database.QuerySelectColumns
uses gw.api.path.Paths
uses gw.pl.persistence.core.Key

// Get the current bundle from the programming context.
var bundle = gw.transaction.Transaction.getCurrent()
var query = Query.make(Address)

var results = query.select({
    QuerySelectColumns.pathWithAlias("CUser", Paths.make(Address#CreateUser)),
    QuerySelectColumns.path(Paths.make(Address#PublicID))
})

for (row in results) {
    if (row.getColumn("CUser") != null) {
        var user = bundle.loadBean(row.getColumn("CUser") as Key)
        print("User login name: " + (user as User).Credential.UserName + " created address: "
            + row.getColumn("Address.PublicID"))
    }
}
```

To run the preceding code in the Gosu Scratchpad, use the method `Transaction.runWithNewBundle` instead of `Transaction.getCurrent`. Enclose the remaining code in a code block, as shown in the following code.

```
gw.transaction.Transaction.runWithNewBundle( \ bundle -> {
    var query = Query.make(Address)
    ...
}, "su")
```

On your production system, Guidewire strongly recommends that you do not use the Guidewire sys user (System User), or any other default user, to create a new bundle.

See also

- “Building a simple query” on page 748

Limitations of row queries

Row queries have the following limitations:

Result values are not part of object domain graphs

You cannot use object path notation with values in results from row queries. The values in results from row queries are not part of object domain graphs.

Access only to columns and foreign keys

You can access only values of columns and foreign keys with row queries. To access values of virtual properties, or properties of entities through arrays or one-to-ones, you must use entity queries, additional queries, or bundles.

Result values are not in the application cache

Values in the result are in local memory only. The application cache does not contain the results.

Results cannot be updated

You cannot update the database with changes that you make to data returned by a row query.

Incompatible with list views or detail panels

The columns in a row in a row query result do not have properties that the ClaimCenter user interface can access. You must transform the row query results to another type before using them as data sources for list views or detail panels in page configuration files.

See also

- “Transforming results of row queries to other types” on page 785
- “Comparison of entity and row queries” on page 789
- “Performance differences between entity and row queries” on page 801

Working with results

The reason to build queries is to use the information they return to your Gosu program. Frequently you use items returned in result objects to display information in the user interface. For example, you might query the database for a list of doctors and display their names in a list. If you expect that the query will return more results than the user interface can display in a single page, you can set the page size to prefetch query results. Setting the page size is also useful if you want to use only the first few rows of a result set.

See also

- “Setting the page size for prefetching query results” on page 804

What result objects contain

The query builder API supports two types of queries:

- **Entity queries** – Result objects contain a list of references to instances of the primary entity type for the query. You set up an entity query with the `select` method that takes no arguments. For example:

```
query.select()
```

- **Row queries** – Result objects contain a set of row-like structures with values fetched from or computed by the relational database. You set up a row query by passing a Gosu array of column selections to the `select` method. For example:

```
query.select({
    QuerySelectColumns.path(Paths.make(Person#Subtype)),
    QuerySelectColumns.pathWithAlias("FName", Paths.make(Person#FirstName)),
    QuerySelectColumns.pathWithAlias("LName", Paths.make(Person#LastName))
})
```

Contents of result sets from entity queries

Result objects from entity queries contain a list of references to instances of the primary entity type for the query. For example, you write the following Gosu code to begin a new query.

```
var query = Query.make(Company)
```

The preceding sample code sets the primary entity type for the query to `Company`. Regardless of related entity types that you join to the primary entity type, the result contains only instances of the primary entity type, `Company`.

To set up the preceding query as an entity query, call the `select` method without parameters, as the following Gosu code shows.

```
var entityResult = query.select() // The select method with no arguments sets up an entity query.
```

The members of results from entity queries are instances of the type from which you make queries. In this example, the members of `entityResult` are instances of `Company`. After you retrieve members from the results of entity queries, use object path notation to access objects, methods, and properties from anywhere in the object graph of the retrieved member.

The following Gosu code prints the name and the city of the primary address for each Company in the result. The `Name` property is on a `Company` instance, while the `City` property is on an `Address` instance.

```
for (company in entityResult) { // Members of a result from entity query are entity instances.
    print (company.Name + ", " + company.PrimaryAddress.City)
}
```

Contents of result sets from row queries

Result objects from row queries contain a set of row-like structures that correspond to the column selection array that you pass as an argument to the `select` method. Each structure in the set represents a row in the database result set. The members of each structure contain the values for that row in the database result set.

For example, you write the following Gosu code to begin a new query.

```
var query = Query.make(Company)
```

The preceding sample code sets the primary entity type for the query to `Company`. You can join other entity types to the query and specify restrictions, just like you can with entity queries. Unlike entity queries however, the results of row queries do not contain entity instances.

The results of row queries contain values selected and computed by the relational database, based on the column selection array that you pass to the `select` method. To set up the preceding query as a row query, call the `select` method and pass a Gosu array that specifies the columns you want to select for the result.

```
uses gw.api.database.Query
uses gw.api.database.QuerySelectColumns
uses gw.api.path.Paths

var query = Query.make(Company)
var rowResult = query.select({ // The select method with an array argument sets up a row query.
    QuerySelectColumns.pathWithAlias("Name", Paths.make(Company#Name)),
    QuerySelectColumns.pathWithAlias("City", Paths.make(Company#PrimaryAddress, Address#City)),
    QuerySelectColumns.path(Paths.make(Company#PrimaryAddress, Address#Country))
})
```

The members of results from row queries are structures that correspond to the column selection array that you specify.

After you retrieve a member from the result of a row query, you can only use the values the member contains. You cannot use object path notation with a retrieved member to access objects, methods, or properties from the domain graph of the primary entity type of the query.

The following Gosu code prints the company name and the city of the primary address for each `Company` in the result. The result member provides the `Name` column and the `City` column.

```
for (company in rowResult) { // Members of a result from a row query are QueryRow objects.
    print (company.getColumn("Name") + ", " +
        company.getColumn("City") + ", " +
        company.getColumn(2))
}
```

You can only access columns from the members of a row result if you specified them in the column selection array.

See also

- “Paths” on page 808

Comparison of entity and row queries

The following table compares features of entity queries and row queries.

Feature	Entity queries	Row queries
Result contents	Results from entity queries contain references to entity instances, so you can use object path notation to access objects, properties, and methods in their object graphs.	Results from row queries contain values with no access to the object graphs of the entity instances that matched the query criteria.

Feature	Entity queries	Row queries
Entity field types	Entity queries can access data from columns, foreign keys, arrays, virtual properties, one-to-ones, and edge foreign keys.	Row queries can access data only from columns and foreign keys.
Application cache	Entity instances in the result are loaded into the application cache.	Values in the result are in local memory only. They are not loaded into the application cache.
Writable results	Entity instances returned in results can be changed in the database by moving them from the result to a writable bundle.	Results cannot be used directly to write changes to the database.
Page configuration	Results can be data sources for list views and detail panels in page configuration.	Results cannot directly be data sources for list views or detail panels in page configuration.

See also

- “Performance differences between entity and row queries” on page 801

Filtering results with standard query filters

Sometimes you want to use a query result object and filter the items in different ways when you iterate the result. For example, you have a query with complicated joins, predicates, and subselects. Several routines need to use the results of this complex query, but each routine processes different subsets of the results. In such cases, you can separate the query from the filtering of its results. You write query builder code in your main routine to construct and refine the query, and then produce a result object with the `select` method. Building the query once in your main routine improves the process of debugging and maintaining the query logic.

After you obtain a result object in your main routine, pass the result object to each subroutine. The subroutines apply their own, more restrictive predicates by using standard query filters. Also, the subroutines can apply their own ordering and grouping requirements. ClaimCenter combines the query predicates from the main routine and the standard query filter predicates from the subroutine to form the SQL query that it sends to the relational database.

In addition to using standard query filters in Gosu code, you can use standard query filters in the page configuration of your ClaimCenter application. List views support a special type of toolbar widget, a toolbar filter. A toolbar filter lets users choose from a drop-down menu of filters to apply to the set of data that a list view contains. Toolbar filters accept different kinds of filters, including standard query filters.

See also

- “Using standard query filters in toolbar filters” on page 793

Creating a standard query filter

A *standard query filter* represents a named query predicate that you can add to a query builder result object. You create a standard query filter by instantiating a new `StandardQueryFilter` object. The `gw.api.filters` package contains the `StandardQueryFilter` class. Its constructor takes two arguments:

- **Name** – A String to use as an identifier for the filter.
- **Predicate** – A Gosu block with a query builder predicate method. You must apply the predicate method to a field in the query result that you want to filter. You can use the same predicate methods in standard query filter predicates that you use on queries themselves.

The following Gosu code creates a standard query filter that can apply to query results that include `Address` instances. The standard query filter predicate uses a `compare` predicate method on the `City` field.

```
var myQueryFilter = new StandardQueryFilter("myQueryFilter",
    \ query -> {query.compare("City", Equals, "Bloomington")})
```

Note: The package `gw.api.filters` contains predefined standard query filters that you can apply as needed.

Adding a standard query filter to a query result

Use the `addFilter` method on a query builder result object to restrict the items you obtain when you iterate the result. The method takes as its single argument a `StandardQueryFilter` object. You can add as many filters as you want to a query result.

The following Gosu code adds a standard query filter to restrict a result object to addresses in the city of Chicago.

```
uses gw.api.database.Query
uses gw.api.filters.StandardQueryFilter

// Create a query of addresses.
var result = Query.make(Address).select()

// Create a standard query filter for addresses in the city of Chicago.
var queryFilterChicago = new StandardQueryFilter("Chicago Addresses",
    \ query -> {query.compare("City", Equals, "Chicago")})

// Add the Chicago addresses filter to the result.
result.addFilter(queryFilterChicago)

// Iterate the addresses in Chicago.
for (address in result) {
    print (address.City + ", " + address.State + " " + address.PostalCode)
}
```

Use the `addFilter` method when you need a single result to have one or more query filters in effect at the same time.

Using AND and OR logic with standard query filter predicates

You can use AND and OR logic with standard query filter predicates in the same way that you use AND and OR logic with predicates on a query builder query.

- **For AND logic** – Add multiple standard query filters, each with a single comparison predicate, to a result.
- **For OR logic** – Add a single standard query filter with an `or` method in its filter predicate to a result.

Using AND logic with standard query filter predicates

With standard query filters, you combine filter predicates that all must be true by adding standard query filters to a result one after the other. The following Gosu code adds two standard query filters to a result object. The first filter restricts the iteration to addresses in the city of Chicago. The second filter further restricts the iteration to addresses that were added to the database during 2017 or later.

```
uses gw.api.database.Query
uses gw.api.filters.StandardQueryFilter
uses gw.api.util.DateUtil

// Create a query of addresses.
var result = Query.make(Address).select()

// Create a standard query filter for addresses in the city of Chicago.
var queryFilterChicago = new StandardQueryFilter("Chicago Addresses",
    \ query -> {query.compare(Address#City, Equals, "Chicago")})

// Create a standard query filter for addresses added during 2017 or later.
var queryFilter2017Address = new StandardQueryFilter("2017 Addresses",
    \ query -> {query.compare(Address#createTime, GreaterThanOrEqual,
        DateUtil.createDateInstance(1, 1, 2017))})

// Add the Chicago and 2017 address filters to the result.
result.addFilter(queryFilterChicago)
result.addFilter(queryFilter2017Address)

// Iterate the addresses in Chicago added during 2016 or later.
for (address in result) {
    print(address.City + ", " + address.State + " " + address.PostalCode + " " + address.CreateTime)
}
```

If you add more than one standard query filter to a result, make sure that each filter predicate applies to a different field in the result. If you add two or more standard query filters with predicates on the same field, Boolean logic ensures that no item in the result satisfies them all.

Using OR logic with standard query filter predicates

With standard query filters, you combine predicates only one of which must be true by using the `or` method in the filter predicate of a single standard query filter. The following Gosu code adds two predicates to a standard query filter to restrict the iteration of addresses to the cities of Bloomington or Chicago.

```

uses gw.api.database.Query
uses gw.api.filters.StandardQueryFilter
uses gw.api.database.QuerySelectColumns
uses gw.api.path.Paths

// Create a query of addresses.
var result = Query.make(Address).select()

// Create a standard query filter for addresses in the cities of Bloomington or Chicago.
var queryFilterBloomingtonOrChicago = new StandardQueryFilter("Bloomington and Chicago Addresses",
    \ query -> {query.or( \ orCriteria -> {
        orCriteria.compare(Address#City, Equals, "Bloomington")
        orCriteria.compare(Address#City, Equals, "Chicago")
    })
})

// Add the Bloomington or Chicago filter to the result.
result.addFilter(queryFilterBloomingtonOrChicago)

// Iterate the addresses in Bloomington or Chicago, in order by city.
for (address in result.orderBy(QuerySelectColumns.path(Paths.make(Address#City)))) {
    print(address.City + ", " + address.State + " " + address.PostalCode)
}

```

See also

- “Combining predicates with AND and OR logic” on page 763

Using standard query filters in Gosu code

You often use standard query filters in Gosu code to separate code that constructs a general purpose query from code in subroutines that process different subsets of the result. Query builder code that constructs a query can be complex if joins, subselects, and compound predicate expressions are involved. Placing this part of your query builder code in one location improves the process of debugging and maintaining your code.

The following Gosu code builds a query for addresses in the state of Illinois, and then passes the query result to three subroutines for processing. Each subroutine creates and adds a standard query filter for a city in Illinois: Bloomington, Chicago, or Evanston. Then, the subroutines iterate their own subset of the main query result.

```

uses gw.api.database.Query
uses gw.api.database.IQueryBeanResult
uses gw.api.filters.StandardQueryFilter
uses gw.api.database.QuerySelectColumns
uses gw.api.path.Paths

// Select addresses from the state of Illinois.
var query = Query.make(Address)
query.compare(Address#State, Equals, typekey.State.TC_IL)

// Get a result and apply ordering
var result = query.select()
result.orderBy(QuerySelectColumns.path(Paths.make(Address#City)))
    .thenBy(QuerySelectColumns.path(Paths.make(Address#PostalCode)))

// Pass the ordered result to subroutines to process Illinois addresses by city.
processBloomington(result)
result.clearFilters() // Remove any filters added by the subroutine from the result.

processChicago(result)
result.clearFilters() // Remove any filters added by the subroutine from the result.

processEvanston(result)
result.clearFilters() // Remove any filters added by the subroutine from the result.

// Subroutines

// Add Bloomington filter and process results.
function processBloomington(aResult : IQueryBeanResult<Address>) {
    var queryFilterBloomington = new StandardQueryFilter("QueryFilterBloomington",
        \ filterQuery -> {filterQuery.compare(Address#City, Equals, "Bloomington")})
    aResult.addFilter(queryFilterBloomington)
    for (address in aResult) {
        print (address.City + ", " + address.State + " " + address.PostalCode)
    }
}

```

```
}

// Add Chicago filter and process results.
function processChicago(aResult : IQueryBeanResult<Address>) {
    var queryFilterChicago = new StandardQueryFilter("QueryFilterChicago",
        \ filterQuery -> {filterQuery.compare(Address#City, Equals, "Chicago")})
    aResult.addFilter(queryFilterChicago)
    for (address in result) {
        print (address.City + ", " + address.State + " " + address.PostalCode)
    }
}

// Add Evanston filter and process results.
function processEvanston(aResult : IQueryBeanResult<Address>) {
    var queryFilterEvanston = new StandardQueryFilter("QueryFilterEvanston",
        \ filterQuery -> {filterQuery.compare(Address#City, Equals, "Evanston")})
    aResult.addFilter(queryFilterEvanston)
    for (address in result) {
        print (address.City + ", " + address.State + " " + address.PostalCode)
    }
}
```

Using standard query filters in toolbar filters

You often use standard query filters with list views in the page configuration of the application user interface. The row iterators of list views support toolbar filter widgets. A toolbar filter lets users select from a drop-down menu of query filters to view subsets of the data that the list view displays. Standard query filters are one type of query filter that you can add.

You specify the standard query filters for a toolbar filter on the **Filter Options** tab. On the tab, you can add two kinds of filter options:

- **ToolbarFilterOption** – An expression that resolves to a single object that implements the `BeanBasedQueryFilter` interface, such as a standard query filter.
- **ToolbarFilterOptionsGroup** – An expression that resolves to an array of objects that implement the `BeanBasedQueryFilter` interface, such as standard query filters.

You can specify a standard query filter or an array of standard query filters by using an inline constructor in the `filter` property of a filter option. Alternatively, you can specify a Java or Gosu class that returns a standard query filter or an array of them.

Toolbar filter caching

Typically, list views cache the most recent toolbar filter selection made in a user's session. If the user leaves a page and then returns to it, a list view retains and applies the toolbar filter option that was in effect when the user left the page.

You disable filter caching by setting the `cacheKey` property of a toolbar filter to an expression that evaluates to a potentially different value each time a user enters the page. For example, you might specify the following Gosu expression.

```
claim.ClaimNumber
```

If you disable filter caching, the list view reverts to the default filter option for entry to the page. You specify the default filter option by setting the `selectOnEntry` property on the option to `true`. Alternatively, you specify the default filter option by moving the option to the top of the list of options on the **Filter Options** tab.

Toolbar filter recalculation

Generally, list views calculate their filter options only once, when a user enters a page. Filter options remain unchanged during the life span of a page. Sometimes you need a list view to recalculate its filter options in response to changes that a user makes on a page.

You force a list view to recalculate its filter options in response to changes by setting the `cacheOptions` property of a toolbar filter to `false`. Set the property to `false` with caution, because the recalculation of filter options after a user makes changes can reduce the speed at which the application renders the updated page.

Example of a single toolbar filter option

The following example `filter` properties each specify a standard query filter by using an inline constructor. The filter applies to work queue tasks that the list view in the `WorkQueueExecutorsPanelSet` PCF file displays.

`WorkQueueExecutorsPanelSet` is a separate PCF file included in `WorkQueueInfo.pcf`. The `TaskFilter` toolbar filter has two `ToolbarFilterOption` filter values:

```
new gw.api.filters.StandardQueryFilter("All", \ q -> {})
new gw.api.filters.StandardQueryFilter("With errors",
    \ q -> q.compare("Exceptions", gw.api.database.Relop.GreaterThan, 0))
```

Note: The previous code block for the single-line `filter` option field in the PCF editor contains line breaks and extra spaces for readability. If you copy and paste this code, remove these line breaks and spaces to make the code valid.

The toolbar filter uses the first parameters of the filters to provide two options, “All” and “With errors”, in the toolbar filter drop-down menu.

For single filter options, you can override the text of the drop-down menu of with the `label` property. For localization purposes, you must specify the filter name or the `label` property as a display key, not as a `String` literal.

Example of a group toolbar filter option

The following example `filters` property combines the two `filter` properties in the `WorkQueueExecutorsPanelSet` PCF file to specify an array of two standard query filters by using inline constructors. This `ToolbarFilterOptionGroup` example replaces the two `ToolbarFilterOption` filter values on the `TaskFilter` toolbar filter:

```
new gw.api.filters.StandardQueryFilter[] {
    new gw.api.filters.StandardQueryFilter("All", \ q -> {}),
    new gw.api.filters.StandardQueryFilter("With errors",
        \ q -> q.compare("Exceptions", gw.api.database.Relop.GreaterThan, 0)))
```

Note: The previous code block for the single-line `filters` option field in the PCF editor contains line breaks and extra spaces for readability. If you copy and paste this code, remove these line breaks and spaces to make the code valid.

The toolbar filter displays the first parameters of the filters, “All” and “With errors”, as options in the toolbar filter drop-down menu. The drop-down menu displays them together and in the order that you specify in the array constructor.

Group filter options do not have a `label` property, so the text of the menu options comes only from the filter names. For localization purposes, you must specify the filter names as display keys, not as `String` literals.

Ordering results

By default, SQL Select statements and the query builder APIs return results from the database in no specific order. Results in this apparently random order might not be useful. SQL and the query builder APIs support specifying the order of items in the results.

With SQL, the `ORDER BY` clause specifies how the database sorts fetched data. The following SQL statement sorts the result set on the postal codes of the addresses.

```
SELECT * FROM addresses
WHERE city = "Chicago"
ORDER BY postal_code;
```

The following Gosu code uses the `orderBy` ordering method to sort addresses in the same way as the preceding SQL example.

```
uses gw.api.database.Query
uses gw.api.database.QuerySelectColumns
uses gw.api.path.Paths

var queryAddresses = Query.make(Address) // Query for addresses in Chicago.
queryAddresses.compare(Address#City, Equals, "Chicago")
```

```
var selectAddresses = queryAddresses.select()  
// Sort the result by postal code.  
selectAddresses.orderBy(QuerySelectColumns.path(Paths.make(Address#PostalCode)))  
var resultAddresses = selectAddresses.iterator() // Execute the query and iterate the ordered results.
```

The query builder APIs use the object path expressions that you pass to ordering methods to generate the ORDER BY clause for the SQL query. Gosu does not submit the query to the database until you begin to iterate a result object. The database fetches the items that match the query predicates, sorts the fetched items according to the ordering methods, and returns the result items in that order.

Ordering methods of the query builder API

The Query builder API supports the following ordering methods on result objects.

Method	Description
orderBy	Clears all previous ordering, and then orders results by the specified column in ascending order.
orderByDescending	Clears all previous ordering, and then orders results by the specified column in descending order.
thenBy	Orders by the specified column in ascending order, without clearing previous ordering.
thenByDescending	Orders by the specified column in descending order, without clearing previous ordering.

The ordering methods all take an object that implements the `IQuerySelectColumn` interface in the `gw.api.database` package as their one argument. To create this object, use the following syntax.

```
QuerySelectColumns.path(Paths.make(PrimaryEntity#SimpleProperty))
```

The `Paths.make` method yields an object access path from the primary entity to a simple, non-foreign-key, database-backed property. Gosu checks property names against column names in the *Data Dictionary*. To specify a path to a property on a dependent table, specify a foreign-key property to the dependent table on the previous table parameter.

For example, the following Gosu code specifies a simple property, the `PostalCode` on an `Address` instance.

```
QuerySelectColumns.path(Paths.make(Address#PostalCode))
```

See also

- “Paths” on page 808
- “Locale sensitivity for ordering query results” on page 796
- Globalization Guide*

Ordering query results on related instance properties

The ordering methods of the query builder APIs let you order results on the properties of related entities. To do so, specify an object access path from the primary entity of the query. The access path can traverse only database-backed foreign keys, and must end with a simple, database-backed property. The access path cannot include virtual properties, methods, or calculations. You do not need to join the related entities to your query to reference them in the parameters that you pass to the ordering methods.

The following Gosu code orders notes, based on the date that the activity related to a note was last viewed.

```
uses gw.api.database.Query  
uses gw.api.database.QuerySelectColumns  
uses gw.api.path.Paths  
  
var queryNotes = Query.make(Note) // Query for notes.  
var resultNotes = queryNotes.select()  
  
// Sort the notes by related date on activity.  
resultNotes.orderBy(QuerySelectColumns.path(Paths.make(Note#Activity, Activity#LastViewedDate)))
```

Multiple levels of ordering query results

Often you need to order results on more than one column or property. For example, you query addresses and want them ordered by city. Within a city, you want them ordered by postal code. Within a postal code, you want them ordered by street. In this example, you want three levels of ordering: city, postal code, and street.

The following SQL statement specifies three levels of ordering for addresses.

```
SELECT * FROM addresses
ORDER BY city, postal_code, address_line1;
```

The following Gosu code constructs and executes a query that is functionally equivalent to the preceding SQL example.

```
uses gw.api.database.Query
uses gw.api.database.QuerySelectColumns
uses gw.api.path.Paths

var queryAddresses = Query.make(Address) // Query for addresses.
var resultAddresses = queryAddresses.select()

// Sort results by city.
resultAddresses.orderBy(QuerySelectColumns.path(Paths.make(Address#City)))
// Within a city, sort results by postal code.
resultAddresses.thenBy(QuerySelectColumns.path(Paths.make(Address#PostalCode)))
// Within a postal code, sort results by street.
resultAddresses.thenBy(QuerySelectColumns.path(Paths.make(Address#AddressLine1)))
```

You can call the ordering methods `thenBy` and `thenByDescending` as many times as you need.

Locale sensitivity for ordering query results

The query builder APIs order query results by using locale-sensitive comparisons. In contrast, collection enhancement methods for ordering rely on comparison methods built into the Java interface `java.lang.Comparable`. To sort `String` values in a locale-sensitive way, you supply an optional `Comparator` argument to those methods.

Note: In configurations that use primary or secondary linguistic sort strength, the ordering of results is case-insensitive. If either of your ClaimCenter or the relational database that your ClaimCenter uses have these configurations, the query result includes ignores the case of `String` values. For example, in these configurations, “Building Renovators” and “building renovators” are adjacent in the results.

See also

- *Gosu Reference Guide*
- *Globalization Guide*

Useful properties and methods on result objects

Query result objects have these useful properties:

Empty

Indicates whether a query failed to fetch any matching items

Count

Returns the number of items in the result

AtMostOneRow

Specifies that you want the query to return a result only if a single item matches the query predicates

```
var uniquePerson = query.select().AtMostOneRow
```

FirstResult

Provides the first item in a result without iterating the result object

getCountLimitedBy

Indicates whether the result contains more items than you can use

Determining whether a query returned no results

Queries can return no results under certain conditions, such as queries that include predicates that users provide through the user interface. For example, users can search for people by name by using the user interface, such as finding people whose name is “John Smith.” The database could have several matches, or it could have none.

Your user interface displays a list of names when there is a match or the message “No-one found with that name” when there is none. Use the `Empty` property on a result object to determine whether a query returned any results, as the following Gosu code shows.

```
uses gw.api.database.Query  
  
var query = Query.make(Person)  
  
// Apply some subselect, join, and predicate methods here.  
  
result = query.select()  
  
if (result.Empty) {  
    ...  
}
```

Alternatively, you can test the `Count` property against zero or iterate a result object inside a `while` loop with a counter and then test the counter for zero. Relational query performance often improves if you use the `Empty` property.

Result counts and dynamic queries

A query is always dynamic and returns results that may change if you use the object again. Some rows may have been added, changed, or removed from the database from one use of the query object to another use of the query, even within the same function.

The `Count` property of a query gets the current count of items. Do not rely on the count number remaining constant. That number might be useful in some contexts such as simple user interface messages such as “Displaying items 1-10 out of 236 results”. However, that count might be different from the number of items returned from a query even if you iterate across the result set immediately after retrieving the count. Other actions may add, change, or remove rows from the database between the time you access the `Count` property and when you iterate across the results.

Unsafe usage:

```
// Bad example. Do NOT follow this example. Do NOT rely on the result count staying constant!  
  
uses gw.api.database.Query  
  
// create a query  
var query = Query.make(User)  
  
// THE FOLLOWING LINE IS UNSAFE  
var myArray = new User[ query.select().Count ]  
  
for (user in query.select() index y) {  
    // This line throws an out-of-bounds exception if more results appear after the count calculation  
    myArray[y] = user  
}
```

Code like the previous example risks throwing array-out-of-bounds errors at run time. Adding a test to avoid the exception risks losing records from the result.

Instead, iterate across the set and count upward, appending query result entities to an `ArrayList`.

Safe usage:

```
uses gw.api.database.Query  
uses java.util.ArrayList  
  
var query = Query.make(User)  
  
// Create a new list, and use generics to parameterize it  
var myList = new ArrayList<User>()  
  
for (user in query.select()) {  
    // Add a row to the list
```

```
    myList.add(user)
}
```

Calling `query.select()` does not snapshot the current value of the result set forever. When you access the `query.select().Count` property, ClaimCenter runs the query but the query results can change quickly. Database changes could happen in another thread on the current server or on another server in the cluster.

Accessing the first item in a result

Sometimes you want only the first item in a result, regardless of how many instances the database can provide. Use the `FirstResult` property on a result object to obtain its first item, as the following Gosu code shows.

```
uses gw.api.database.Query
var query = Query.make(Person)
query.compareIgnoreCase(Person#FirstName, Equals, "ray")
query.compareIgnoreCase(Person#LastName, Equals, "newton")
firstPerson = query.select().FirstResult
```

As an alternative, you can iterate a result and stop after retrieving the first item. However, relational query performance often improves when you use the `FirstResult` property to access only the first item in a result.

Note: To find the last item in a result, reverse the order of the rows by calling the `orderByDescending` method.

Determining if a result will return too many items

When you develop search pages for the user interface, you may want to limit the number of results that you display in the list view. For example, if a user provides little or no search criteria, the number of items returned could be overwhelming. The `getCountLimitedBy` method on result objects lets you efficiently determine how many items a result will return, without fetching all the data from the database. If it will return too many items, your search page can prompt the user to narrow the selection by providing more specific criteria.

With the `getCountLimitedBy` method, you specify a threshold. The *threshold* is the number of items that is more than you require, which is the maximum number of items that you want, plus one. If the number of items that the result will contain falls below the threshold, the method returns the actual result count. On the other hand, if the number of items is at or above the threshold, the method returns the threshold, not the actual result count.

For example, you want to limit search results to a hundred items. Pass the number 101 to the `getCountLimitedBy` method, and check the value that the method returns. If the method returns a number less than 101, the result will be within your upper bound, so you can safely iterate the result. If the method returns 101, the result will cross the threshold. In this case, prompt the user to provide more precise search criteria to narrow the result.

The following Gosu code demonstrates how to use the `getCountLimitedBy` method.

```
uses gw.api.database.Query
// Query for addresses in the city of Chicago
var query = Query.make(Address)
query.compare(Address#City, Equals, "Chicago")
var result = query.select()

// Specify the threshold for too many items in the result
var threshold = 101 // -- 101 or higher is too many
result.getCountLimitedBy(threshold)

// Test whether the result count crosses the threshold
if (result.getCountLimitedBy(threshold) < threshold) {
    // Iterate the results
    for (address in result) {
        print (address.AddressLine1 + ", " + address.City + ", " + address.PostalCode)
    }
} else {
    // Prompt for more criteria
    print ("The search will return too many items!")
}
```

Converting result objects to lists, arrays, collections, and sets

You can convert the query result objects of entity queries to lists, arrays, collections, and sets by using the conversion methods:

- `toCollection`
- `toSet`
- `toList`
- `toTypedArray`

IMPORTANT: Do not use the `where` method on lists, arrays, collections, or sets returned by conversion methods on result objects. Instead, apply all selection criteria to query objects by using predicate methods before calling the `select` method.

Converting a query result to these types executes the database query and iterates across the entire set. The application pulls all entities into local memory. Because of limitations of memory, database performance, and CPU performance, never do this conversion for queries of unknown size. Only do this conversion if you are absolutely certain that the result set size and the size of the object graphs are within acceptable limits. Be sure to test your assumptions under production conditions.

WARNING: Converting a query result to a list or array pulls all the entities into local memory. Do this conversion only if you are absolutely certain that the result set size and the size of the objects are small. Otherwise, you risk memory and performance problems.

If you need the results as an array, you can convert to a list and then convert that to an array using Gosu enhancement methods.

The following example converts queries to different collection-related types, including arrays:

```
uses gw.api.database.Query

// Query for addresses in the city of Chicago
var query = Query.make(Address)
query.compare(Address#City, Equals, "Chicago")
var result = query.select()

// Specify the threshold for too many items in the result
var threshold = 101 // -- 101 or higher is too many
result.getCountLimitedBy(threshold)

// Test whether the result count crosses the threshold
if (result.getCountLimitedBy(threshold) < threshold) {
    // Iterate the results
    for (address in result) {
        print(address.AddressLine1 + ", " + address.City + ", " + address.PostalCode)
    }
} else {
    // Prompt for more criteria
    print("The search will return too many items!")
}
var query = Query.make(User)

// In production code, converting to lists or arrays can be dangerous due
// to memory and performance issues. Never use this approach unless you are certain
// the result size is small. In this demonstration, we check the count first.
// In real-world code, you would not use this approach with the User table, which
// can be larger than 100 rows.
if (query.select().Count < 100) {
    var resultCollection = query.select().toCollection()
    var resultSet = query.select().toSet()
    var resultList = query.select().toList()
    var resultArray = query.select().toTypedArray()

    print("Collection: typeof " + typeof resultCollection + " / size " + resultCollection.Count)
    print("Set: typeof " + typeof resultSet + " / size " + resultSet.Count)
    print("List: typeof " + typeof resultList + " / size " + resultList.Count)
    print("Array: typeof " + typeof resultArray + " / size " + resultArray.Count)
} else {
    throw("Too many query results to convert to in-memory collections")
}
```

This example prints something like the following:

```
Collection: typeof java.util.ArrayList/ size 34
Set: typeof java.util.HashSet/ size 34
List: typeof java.util.ArrayList/ size 34
Array: typeof entity.User[]/ size 34
```

See also

- “Transforming results of row queries to other types” on page 785

Updating entity instances in query results

Entities that you iterate across in a query result are read-only by default. The query builder APIs load iterated entities into a read-only bundle. A *bundle* is a collection of entities loaded from the database into server memory that represents a transactional unit of information. The read-only bundles that contain iterated query results are separate from the active read-write bundles of running code. You cannot update the properties of query result entities while they remain in the read-only bundle of the result. In many uses of query results, you need to write changed records to the database. For example, you use custom batch processing to flag contacts that your users need to call the next day.

Moving entities from query results to writable bundles

To change the properties of entities in query results, you must move the entities from the query result to a writable bundle. To move an entity to a writable bundle, call the `add` method on the writable bundle and save the result of the `add` method in a variable. Whenever you pass an entity from a read-only bundle to a writable bundle, the `add` method returns a clone of the entity instance that you passed to the method.

If you move an entity from a query result to a writable bundle, store the entity reference that the `add` method returns in a variable. Then, modify the properties on the saved entity reference. Do not modify properties on the original entity reference that remains in the result set or its iterator. Avoid keeping any references to the original entity instance whenever possible.

Moving entities from query results to the current bundle

Most programming contexts have a writable bundle that the application prepares and manages. For example, all rule execution contexts and PCF code has a current bundle. When a current bundle exists, get it by using the `getCurrent` method.

Typically, whenever you want to update an entity in a query result, you move it to the current bundle. While you iterate the result, add each entity to the current bundle and make changes to the version of it that the `add` method returns. After you finish iterating the query result and the execution context finishes, the application commits all the changes that you made to the database.

IMPORTANT: Entities must not exist in more than one writable bundle at a time.

For example:

```
// Get the current bundle from the programming context.
var bundle = gw.transaction.Transaction.getCurrent()

var query = gw.api.database.Query.make(Address)
query.compare(Address#State, Equals, typekey.State.TC_IL)
var result = query.select().orderBy(QuerySelectColumns.path(Paths.make(Address#City)))

for (address in result) {
    switch (address.City) {
        case "Schaumburg":
        case "Melrose Park":
        case "Norridge":
            break
        default: {
            // Add the address to the current bundle to make it writable and
            // discard the read-only reference obtained from the query result.
            address = bundle.add(address)
            // Change properties on the writable address.
            address.Description =
                // Use a Gosu string template to concatenate the current description and the new text.
                "${address.Description}; This city is not Schaumburg, Melrose Park, or Norridge."
        }
    }
}
```

```
    }  
}
```

At the time the execution context for the preceding code finishes, the application commits all changes made to addresses in the current bundle to the database.

The Gosu Scratchpad does not have a current bundle, so you must create a new bundle. To run the preceding code in the Gosu Scratchpad, use the method `Transaction.runWithNewBundle` instead of `Transaction.getCurrent`. Enclose the remaining code in a code block:

```
gw.transaction.Transaction.runWithNewBundle( \ bundle -> {  
    var query = gw.api.database.Query.make(Address)  
    ...  
}, "su")
```

To see the effects of the preceding code, run the following Gosu code.

```
// Query the database for addresses in Illinois.  
var query = gw.api.database.Query.make(Address)  
query.compare(Address#State, Equals, typekey.State.TC_IL)  
  
// Configure the result to be ordered by City.  
var result = query.select()  
result.orderBy(QuerySelectColumns.path(Paths.make(Address#City)))  
  
// Iterate and print the result set.  
for (address in result) {  
    print(address.City + ", " + address.Description)  
}
```

See also

- “`setChunkingById`” on page 730
- *Gosu Reference Guide*

Testing and optimizing queries

Performance differences between entity and row queries

The query builder API supports two types of queries:

- **Entity queries** – Result objects contain a list of references to instances of the primary entity type for the query. You set up an entity query with the `select` method that takes no arguments. For example:

```
query.select()
```

- **Row queries** – Result objects contain a set of row-like structures with values fetched from or computed by the relational database. You set up a row query by passing a Gosu array of column selections to the `select` method. For example:

```
query.select({  
    QuerySelectColumns.path(Paths.make(Person#Subtype)),  
    QuerySelectColumns.pathWithAlias("FName", Paths.make(Person#FirstName)),  
    QuerySelectColumns.pathWithAlias("LName", Paths.make(Person#LastName))  
})
```

Some situations require entity queries. For example, in page configuration, you must use entity queries as the data sources for list views and detail panels. Other situations require row queries. For example, you must use row queries to produce the results of SQL aggregate queries or outer joins. In yet other cases, you can use either type of query. For example, you can use either entity or row queries in Gosu code used in batch processing types.

Use entity queries for code readability and maintenance

Generally, use an entity query unless performance is not adequate. The query builder code for entity queries is more readable and maintainable than for row queries, especially if you access the query builder APIs from Java rather than Gosu.

Use row queries to improve performance

Generally, use a row query instead of an entity query if the performance of the entity query is inadequate. Entity queries sometimes fetch unused data columns or execute additional queries in code that processes the results. Row queries can improve performance in these cases. Navigating object path expressions that span arrays and foreign keys to access the data that you need can cause additional, implicit database queries to fetch the data. Row queries fetch only the data that you need and typically avoid these additional queries.

Row queries can select fields

You can reduce the number of queries implicitly executed by row queries if you select specific fields rather than entity instances. If you specify fields rather than instances, the relational database fetches and computes values in response to the SQL query that the query builder expression submits to the database.

See also

- “Limitations of row queries” on page 787

Viewing the SQL select statement for a query

The query builder APIs provide two ways to preview and record SQL SELECT statements that your Gosu code and the query builder APIs submit to the application database:

toString

Provides an approximation of the SQL Select statement before it is submitted

withLogSQL

Displays and records the exact SQL Select statement at the time it is submitted

Using `toString` to preview SQL SELECT statements

You may want to see what the underlying SQL SELECT statement looks like as you build up your query. Use the Gosu `toString` method to return an approximation of the SQL SELECT statement at given points in your Gosu code. That way you can learn how different query builder methods affect the underlying SQL SELECT statement.

For example, the following Gosu code prints the SQL Select statement as it exists after creating a query.

```
uses gw.api.database.Query
var query = Query.make(Contact)
print(query.toString())
```

The output looks like the following:

```
[]  
SELECT /* cc:T:Gosu class redefiner; */ FROM cc_contact gRoot WHERE gRoot.Retired = 0
```

The first line shows square brackets ([]) containing the list of variables to bind to the query. In this case, there are no variables. The remaining lines show the SQL statement. Note that the table for the entity type, `cc_contact`, has the table alias `gRoot`.

After you join a related entity to a query or apply a predicate, use the `toString` method to see what the query builder APIs added to the underlying SQL statement.

Note: The `toString` method returns only an approximation of the SQL statement that ClaimCenter submits to the application database. The actual SQL statement might differ due to database optimizations that ClaimCenter applies internally.

Using `withLogSQL` to record SQL SELECT statements

You might want to see the underlying SQL SELECT statement that the query builder API submits to the application database. Use the `withLogSQL` method on a query to see the statement. When you turn on logging behavior, the query builder API writes the SQL SELECT statement to the system logs, in logging category `Server.Database`. The API also writes the SQL statement to standard output (`stdout`).

For example, the following Gosu code turns logging on by calling the `withLogSQL` method of the query object. The SQL statement is written to the system logs at the time code starts to iterate the results.

```
uses gw.api.database.Query

var query = Query.make(Person).withLogSQL(true)
query.startsWith(Person#LastName, "A", false)
query.withLogSQL(true)           // -- turn on logging behavior here --
var result = query.select().orderBy(QuerySelectColumns.path(Paths.make(Person#LastName)))
    .thenBy(QuerySelectColumns.path(Paths.make(Person#FirstName)))

var i = result.iterator()       // -- write the SQL statement to the system logs here --
while (i.hasNext()) {
    var person = i.next()
    print (person.LastName + ", " + person.FirstName + ": " + person.EmailAddress1)
}
```

The SQL Select statement for the preceding example looks like the following on standard output.

```
Executing sql = SELECT /* KeyTable:cc_contact; */
gRoot.ID col0, gRoot.Subtype col1, gRoot.LastName col2, gRoot.FirstName col3
FROM cc_contact gRoot
WHERE gRoot.Subtype IN (?, ?, ?, ?, ?, ?) AND gRoot.LastName LIKE ? AND gRoot.Retired = 0
ORDER BY col2 ASC, col3 ASC
[1 (typekey), 4 (typekey), 5 (typekey), 9 (typekey), 13 (typekey), 15 (typekey), A% (lastname)]
```

The statement on your system depends on your relational database and might differ from the preceding example.

Note: Writing to the system logs and to standard output does not occur when Gosu code calls the `withLogSQL` method. That logging occurs some time later, when ClaimCenter submits the query to the relational database.

Enabling context comments in queries on SQL Server

On SQL Server, tuning queries can be difficult because you cannot determine what part of the application generates specific queries. To help tune certain queries, you can enable two configuration parameters in `config.xml`.

- `IdentifyQueryBuilderViaComments` – Instrumentation comments from higher level database objects constructed by using the query builder APIs
- `IdentifyORMLayerViaComments` – Instrumentation comments from lower level objects, such as beans, typelists, and other database building blocks

If you set either parameter to true, ClaimCenter adds SQL comments with contextual information to certain SQL SELECT statements that it sends to the relational database. The default for `IdentifyQueryBuilderViaComments` is true. The default for `IdentifyORMLayerViaComments` is false.

The SQL comments are in the format:

```
/* applicationName:ProfilerEntryPoint */
```

The `applicationName` component of the comment is `ClaimCenter`.

The `ProfilerEntryPoint` component of the comment is the name of an entry point known to the Guidewire profiler for that application. For example, `ProfilerEntryPoint` might have the value `WebReq:ClaimSearch`.

Including retired entities in query results

When an entity instance is retired, its `Retired` property is set to `true`. The Data Dictionary tells you which entity types can be retired by including `Retireable` in their lists of delegates. By default, query results do not include retired instances, even if they satisfy all the predicates of the query. For query purposes generally, you must treat retired entities as if they were deleted and no longer in the application database.

To include retired instances in the results of a query, call the `withFindRetired` method with the value `true`, as the following Gosu code shows.

```
uses gw.api.database.Query

var query = Query.make(Activity)
```

```
query.compare(Activity#ActivityPattern, Equals, null)
query.withFindRetired(true)
```

IMPORTANT: Use the `withFindRetired` method on queries only under exceptional circumstances that require exposing retired entities.

Setting the page size for prefetching query results

When you first begin to iterate a query result, the query builder APIs submit the query to the database. Gosu does not typically submit the query each time you access the next result in the result object. Instead, Gosu automatically loads several results from the database result set as a batch into a cache in the application server for quick access. Common actions like iterating across the query have higher performance by using this cache.

In SQL, this caching is known as prefetching results. The number of items that the database prefetches and gives to an application is known as the *prefetch page size*.

Using only the first page of the results is equivalent to the SQL `TOP` or `LIMIT` functionality.

You can customize the number of results that the query builder APIs prefetch in order to tune overall query performance from the point-of-view the application server. To set the page size, call the `setPageSize` method on the query result object.

For example:

```
uses gw.api.database.Query
var query = Query.make(User)
// -- prefetch 10 entity instances at one time --
query.select().setPageSize(10)
```

Other notes:

- If you plan to modify the entities, you must use a transaction.
- In production code, you must not retrieve too many items and keep references to them. Memory errors and performance problems can occur. Design your code to limit the result set that your code returns.

IMPORTANT: Always test database performance under realistic production conditions before and after changing any performance tuning settings.

See also

- “Updating entity instances in query results” on page 800

Using settings for case-sensitivity of text comparisons

Upper and lower case lettering affect comparison of written English values during query selection and result ordering. Often, you want to ignore the difference between “A” and “a” when queries select text values.

Languages other than English have other character modifications that affect comparison for selection and ordering. For example, many written European languages have characters with accents and other diacritical marks on a base letter. Sometimes you want to ignore the differences between base letters with and without diacritics, such as the differences between “A”, “À”, and “Á”. Asian languages have other concerns, such as single-byte and double-byte characters or between katakana and hirigani characters in Japanese.

The result of comparing the value of a character field with another character value differs depending on the language, search collation strength, and database that your ClaimCenter uses. For example, case-insensitive comparisons produce the same results as case-sensitive comparisons if ClaimCenter has a linguistic search strength of primary.

Some query API methods support specifying whether to ignore differences between letter case. When you specify that you want to ignore case, ClaimCenter uses your localization settings to control the results. For each locale configured

in your ClaimCenter instance, you specify which dimensions of difference you want to ignore. For example, you can configure a locale to ignore case only, to ignore case and accents, or to ignore other dimensions of difference that are relevant to the locale.

Query performance can suffer when you ignore case in query predicate methods. To improve query performance if you typically need to ignore case in text comparisons, set the `SupportsLinguisticSearch` attribute on `column` elements in entity definitions. For a column that has the `SupportsLinguisticSearch` set, ClaimCenter creates a corresponding denormalization column in addition to the standard column that stores the field values. When ClaimCenter stores a value in the standard column, ClaimCenter also stores a value in the denormalization column. ClaimCenter saves a denormalized value by converting the regular value to one that ignores character differences, based on the dimensions of difference that you specified in your localization settings. When a query applies a predicate that specifies ignoring the case of a field, ClaimCenter uses the same algorithm that converted values to store in the denormalization column. The relational database compares the converted bound value to the values in the denormalization column, not the values in the standard column.

For example, you specify the following query:

```
uses gw.api.database.Query  
  
// Query the Person instances for a specific last name.  
var query = Query.make(Person).withLogSQL(true)  
query.compare(Person#LastName, Equals, "smith")
```

Print the rows that the query returns by adding the following lines:

```
// Fetch the data and print it.  
var result = query.select()  
for (person in result) {  
    print (person.DisplayName)  
}
```

Gosu creates a query that looks like the following one.

```
SELECT /* KeyTable:cc_contact; */  
    gRoot.ID col0,  
    gRoot.Subtype col1  
FROM pc_contact gRoot  
WHERE gRoot.LastName = ?  
AND gRoot.Retired = 0 [smith (lastname)]
```

Depending on your ClaimCenter and database settings, you see either no output or output that looks like:

```
Steve Smith  
Kerry Smith  
Alice Smith  
John Smith  
...
```

To do a case-insensitive text comparison, use the `compareIgnoreCase` method. This method uses the same parameters as the `compare` method. To see the effect on the SQL query of using the `compareIgnoreCase` method, change the comparison line in the code to:

```
query.compareIgnoreCase(Person#LastName, Equals, "smith")
```

Gosu creates a query that looks like the following one.

```
SELECT /* KeyTable:cc_contact; */  
    gRoot.ID col0,  
    gRoot.Subtype col1  
FROM pc_contact gRoot  
WHERE gRoot.LastNameDenorm = ?  
AND gRoot.Retired = 0 [smith (lastname {linguistic=true})]
```

The standard column has values like “Smith” and “Menzies”. The denormalization column has corresponding values like “smith” and “menzies”. The preceding query returns Person entity instances with the last name “Smith” in the standard column, because it compared the bound value “smith” to the value in the denormalization column.

You see output that looks like:

```
Steve Smith  
Kerry Smith  
Alice Smith  
John Smith  
...
```

See also

- [Globalization Guide](#)
- [Configuration Guide](#)

Chaining query builder methods

You can use a coding pattern known as method chaining to write query builder code. With *method chaining*, the object returned by one method becomes the object from which you call the next method in the chain. Method chaining lets you fit all of your query builder code on a single line as a single statement. The object type that a chain of methods returns is the return type of the final method in the chain. If you need to use a particular object type as an argument to a function, you must ensure that the object that the chain returns is the correct type.

The following Gosu code places each query builder API call in a separate statement. The object type of `queryPerson` is `Query<Person>`. The object type of `tableAddress` is `Table<Person>`. The return value of the `contains` method, a `Restriction<Person>` object, is not used. The object type of `result` is `IQueryBeanResult<Person>`.

```
uses gw.api.database.Query  
  
// Query builder API method calls as separate statements  
var queryPerson = Query.make(Person)  
var tableAddress = queryPerson.join(Person#PrimaryAddress)  
tableAddress.contains(Address#AddressLine1, "Lake Ave", true)  
  
var result = queryPerson.select()  
  
// Fetch the data with a for loop  
for (person in result) {  
    print (person + ", " + person.PrimaryAddress.AddressLine1)  
}
```

The following Gosu code is functionally equivalent to the sample code above, but it uses method chaining to condense the separate statements into one. Because the code uses only the value of `result`, an `IQueryBeanResult<Person>` object, none of the intermediate return objects are necessary.

```
uses gw.api.database.Query  
  
// Query builder API method calls chained as a single statement  
var result = Query.make(Person).join(Person#PrimaryAddress).contains(Address#AddressLine1,  
    "Lake Ave", true).select()  
  
// Fetch the data with a for loop  
for (person in result) {  
    print (person + ", " + person.PrimaryAddress.AddressLine1)  
}
```

When you chain methods, the objects that support the calls in the chain often do not appear explicitly in your code. In the example above, the `Query.make` method returns a query object, on which the chained statement calls the `join` method. In turn, the `join` method returns a table object, on which the next method calls the `contain` method. The `contain` method returns a restriction object, which has a `select` method that returns a result object for the built-up query. After the single-line statement completes, the query object is discarded and is no longer accessible to subsequent Gosu code.

Note: Method chaining with the query builder APIs is especially useful for user interface development. Page configuration format (PCF) files for list view panels have a single-line property named `value`, which can be a chained query builder statement that returns a result object.

Working with nested subqueries

A *nested subquery* simplifies a complex query by mapping a set of columns to a pseudo-table. You define the pseudo-table as a query that maps to an alias name in the FROM clause of an SQL query. The parent table in the SQL query joins to the pseudo-table in the same way as it joins to any other secondary table. A typical use of a nested subquery is when you need to access an aggregate value from a secondary table. This advanced feature is not normally needed in production code. Use this feature only if necessary due to the performance implications. For more information about nested subqueries, check the documentation for your database. The Oracle term for a nested subquery is an *inline view*. The SQL Server term for a nested subquery is a *derived table*. If you have questions about the usage of nested subqueries, contact Guidewire Customer Support.

For example, the following SQL query uses a nested subquery to compare the minimum value of a secondary table column with the value of a primary table column:

```
SELECT *
  FROM cc_Address
    INNER JOIN ( SELECT Country, MIN(City) MinCity
                  FROM cc_Address
                    GROUP BY Country ) MinCityAddress
      ON cc_Address.City = MinCityAddress.City;
```

To create a nested subquery, call the `inlineView` method on the query. This method returns a query using the new nested subquery and includes all referenced columns from that query in the `select` statement. The `inlineView` method takes the following arguments:

- `joinColumnOnThisTable` – The name of the join column (`String`).
- `inlineViewQuery` – The query (`Query`). This argument cannot be the result of any `Query` method that returns a `Table`.
- `joinColumnOnViewTable` – The name of the join column on the nested query table (`String`).

The method returns a new query that contains the nested subquery. The return type of the method is a `Table` object.

Predicates on the secondary table can use columns on the secondary table or the primary table.

For example, suppose you create two queries, an inner query and an outer query:

```
var innerQuery = Query.make(Address)
var outerQuery = Query.make(Address).withLogSQL(true)
```

Next, create a nested subquery from the inner query to add the `Address#Country` column to the columns that the outer query returns:

```
var nestedQuery = outerQuery.inlineView("Country", innerQuery, "Country")
```

Next, use the `City` column in a new predicate on the outer query:

```
outerQuery.compare(Address#City, Equals,
  nestedQuery.getColumnRef(DBFunction.Min(Paths.make(Address#City))))
```

Test the code:

```
for (row in outerQuery.select()){
  print(row.DisplayName)
}
```

This code prints the display name of any `Address` entity that has a city that matches the city that is alphabetically first for a particular country. This produces the following SQL statement:

```
SELECT /* KeyTable:pc_address; */ gRoot.ID col0, gRoot.Subtype col1
  FROM pc_address gRoot
    INNER JOIN (
      SELECT address_0.Country col0, MIN(address_0.City) col1
        FROM pc_address address_0 WHERE address_0.Retired = 0
          GROUP BY address_0.Country) address_0
```

```
ON gRoot.Country = address_0.col0
WHERE gRoot.City = address_0.col1 AND gRoot.Retired = 0 []
```

Paths

A path is essentially a type-safe list of property references encapsulated in an object of type `gw.api.path.Path`. Each element of the path must have an owning type that is the same type or subtype of the feature type of the immediately previous element in the path. In other words, starting at element 2, the left side of the literal must match the right side of the literal in the previous element in the path.

From a programming perspective, `Path` is an interface. For use with the query builder APIs, the only relevant implementing class is `PersistentPath`, which contains only persistent property references.

Making a path

To create a path, use the `gw.api.path.Paths` class, which has a static method called `make`. Pass each property reference in order as separate ordered method arguments. The `make` method has method signatures that support paths of length 1, 2, 3, 4, and 5. The `make` method only works with property references that represent persistent properties. A *persistent property* is a property that directly maps to a database field.

The following example creates a path with two property references

```
var path = Paths.make(User#Contact, UserContact#PrimaryAddress)
```

Appending a path

You can append a property with the `append` method:

```
var pathPostalCode = path.append(Address#PostalCode)
```

Converting a path to a list

You can convert a path to an immutable list that implements `java.util.List`:

```
var pathAsList = pathPostalCode.asList()
```

The elements of the list are `Contact`, `PrimaryAddress`, and `PostalCode`.

Getting the leaf value from a path and an object

Given an object and a path, you can get the leaf value (the final and rightmost property in the path) using the `getLeafValue` method:

```
var postalCode = pathPostalCode.getLeafValue(user)
```

You must ensure that no element in the path to the leaf is `null`.

An entity query on the `User` table can use this method to access the leaf values:

```
var query = Query.make(User)
var rowResult = query.select()
for (user in rowResult){
    print(user.DisplayName + " " +
          (user.Contact.PrimaryAddress == null ? "No address" : pathPostalCode.getLeafValue(user)))
}
```

Types and methods in the query builder API

The query builder API in the `gw.api.database` package includes Java classes and interfaces and Gosu enhancements. You access the API in your Gosu code in the same way as any other Gosu API. The query builder API provides types and their methods for building queries and for accessing the rows and columns in the query results. The API also

includes types and methods for constructing method parameters. The context-sensitive editor in Studio provides quick access to all these types and methods.

The Javadoc for the Java classes and interfaces in the `gw.api.database` package is available in `ClaimCenter/javadoc`. You can generate and view the Gosudoc that describes Gosu enhancements to the API by running the `gwb gosudoc` command. This command produces documentation at `ClaimCenter/build/gosudoc/index.html`.

See also

- [Gosu Reference Guide](#)

Types and methods for building queries

The following sections describe the major classes, interfaces, and methods that you use to build queries.

Types for building queries

The following Gosu types in the `gw.api.database` package provide methods for building a query. `Query`, `Restriction`, and `Table` all implement the `ISelectQueryBuilder` interface. `IQueryBeanResult` implements `IQueryResult`.

Type	Description
<code>Query</code>	A class that represents a query that fetches entities or rows from the application database.
<code>Restriction</code>	An interface that represents a Boolean condition or conditions that restricts the set of items that a query fetches from the application database.
<code>Table</code>	An interface that represents an entity type that you add to the query with a join or a subselect.
<code>IQueryBeanResult</code>	An interface that represents the results of a query that fetches entities from the application database. Adding sorting or filters to this item affects the query.
<code>IQueryResult</code>	An interface that represents the results of a query that fetches rows from the application database. Adding sorting or filters to this item affects the query.

Methods on Query type for building queries

The following methods provide fundamental functionality for building queries. Other methods provide filtering to the query. To see information about methods that provide additional functionality, see the Javadoc. You can chain many of these methods to create concise code. For example, you can specify a query on an entity type, join to another entity type, and create the result set object by using a line like the following one:

```
var result = Query.make(Company).join(Company#PrimaryAddress).select()
```

Method	Description	Returned object type
<code>join</code>	Returns a table object with information from several entity types joined together in advanced ways.	<code>Table</code>
<code>make</code>	Static method on the <code>Query</code> class that creates a new query object.	<code>Query</code>
<code>select</code>	Defines the items to fetch from the application database, according to restrictions added to the query. Returns a result object that provides one of the following types of item: <ul style="list-style-type: none">• Items fetched from a single, root entity type• Data rows containing specified fields from one or more entity types	<code>IQueryBeanResult</code> or <code>IQueryResult</code>
<code>subselect</code>	Joins a dependent source. For example: <pre>var outerQuery = Query.make(User) // Returns User Query var innerQuery = Query.make(Note) // Returns Note Query</pre>	<code>Table</code>

Method	Description	Returned object type
	<pre>// Filter the inner query noteQuery.compareIn(Note#Topic, {NoteTopicType.TC_GENERAL, NoteTopicType.TC_LEGAL}) // Filter the outer query by using a subselect userQuery.subselect(User#ID, InOperation.CompareIn, noteQuery, Note#Author)</pre>	
union	Combines all results from two queries into a single result.	GroupingQuery
withDistinct	Whether to remove duplicate entity instances from the result.	Query

Methods on Result types for building queries

The following methods provide ordering functionality for building queries.

Other methods provide filtering to the result set. To see information about methods that provide additional functionality, see the Javadoc.

Method	Description
orderBy	Clears all previous ordering, and then orders results by the specified column in ascending order.
orderByDescending	Clears all previous ordering, and then orders results by the specified column in descending order.
thenBy	Orders by the specified column in ascending order, without clearing previous ordering.
thenByDescending	Orders by the specified column in descending order, without clearing previous ordering.

These ordering methods all take an object that implements the `IQuerySelectColumn` interface as their one argument.

See also

- “Ordering results” on page 794
- “Locale sensitivity for ordering query results” on page 796

Column selection types and methods in the query builder API

The query builder API provides multiple ways to access columns in defining a query and in accessing the column values in the result set. The methods that you use depend on the task that you are performing.

Types for column specification in building a query

The query that you specify by using the `Query.make` method provides access to the properties on the entity type of the query. For example, the following line specifies a query on the `Company` entity type:

```
var queryCompany = Query.make(Company)
```

The `PropertyReference` class provides type-safe access to the properties on the query entity type. You specify a property reference by using the following syntax:

```
EntityType#Property
```

You can use a property reference to filter query results, join to another entity type, order rows in the result set, or specify a column for a row query. For example, the following line joins the `company` query to the `Address` entity type and returns a `Table` object:

```
var tableAddress = queryCompany.join(Company#PrimaryAddress)
```

A `Table` object provides the same type-safe access to properties on its entity type as a `Query` object does. For example, the following lines provide two predicates, one on the query object and one on the table object. These predicates filter

the SQL query that is sent to the database to return only companies with the name “Stewart Media” that have a primary address in Chicago.

```
queryCompany.compare(Company#Name, Equals, "Stewart Media")
tableAddress.compare(Address#City, Equals, "Chicago")
```

If you have a query that joins entity types, the `Query` and the `Table` objects can perform type-safe access only for properties on their own entity type. If you need to compare property values with each other or filter properties on multiple joined tables to create an OR filter, you must use a `ColumnRef` object. For example, the following lines create two predicates on the `Table` object. These predicates filter the SQL query that is sent to the database to return companies with either the name “Stewart Media” or a primary address in Chicago.

```
tableAddress.or( \ or1 -> {
    or1.compare(Address#City, Equals, "Chicago")
    or1.compare(queryCompany.getColumnRef("Name"), Equals, "Stewart Media")
})
```

Some other predicate method signatures take an object that implements the `IQueryablePropertyInfo` interface as a parameter.

Type	Description
PropertyReference	A class that provides a type-safe reference to a property. Use this class to access properties on the primary entity of a <code>Query</code> , <code>Table</code> , or <code>Restriction</code> instance. You can use an instance of this class to create a join or predicate. You can also use an instance of this class to define a query select column to order the query results by a column in a row query. The following lines use property references to join tables and filter the primary entity of a <code>Query</code> and a <code>Table</code> object.
	<pre>var queryCompany = Query.make(Company) var tableAddress = queryCompany.join(Company#PrimaryAddress) queryCompany.compare(Company#Name, Equals, "Stewart Media") tableAddress.compare(Address#City, Equals, "Chicago")</pre>
ColumnRef	A class that specifies a column reference to a property on a dependent entity, or a comparison property on the primary entity of a <code>Query</code> , <code>Table</code> , or <code>Restriction</code> instance. A <code>ColumnRef</code> instance does not provide type safety. The following lines use a column reference for a property on a dependent table:
	<pre>tableAddress.or(\ or1 -> { or1.compare(Address#City, Equals, "Chicago") or1.compare(queryCompany.getColumnRef("Name"), Equals, "Stewart Media") })</pre>
String	Some query builder API method signatures accept a <code>String</code> argument. If a signature that uses a <code>PropertyReference</code> parameter is available, use that type-safe signature instead.
<code>IQueryablePropertyInfo</code>	An interface that provides information about a property. Some query builder API method signatures accept an <code>IQueryablePropertyInfo</code> argument. To create this object, use code like the following:
	<pre>var lastNamesArrayList = {"Smith", "Applegate"} var query = Query.make(Person) query.compareIn(Person.LASTNAME_PROP.get(), lastNamesArrayList)</pre>

Methods for column specification in building a query

You use the following methods on the `QuerySelectColumns` class to specify columns in a type-safe way for row queries and ordering methods on result objects. All the methods return objects of type `IQuerySelectColumn`.

The column selection methods that end with `WithAlias` have a `String` as the first parameter. The value that you pass is an alias for the database column, which becomes the column name in the result of the row query.

The column selection methods that do not end with `WithAlias` have no parameter for column alias in the row query result. To access the column, you must use a numeric position value that starts from 0 or the name of the column as a `String` of format `TABLE.COLUMN_NAME`.

Method	Parameter	Description
pathWithAlias path		<p>Creates a path to a column for ordering query results or to use in a row query. For example, the following lines select columns for a row query and order the results. You use the same syntax for ordering the results of an entity query.</p> <pre>var query = Query.make(Person) var results = query.select({ QuerySelectColumns.pathWithAlias("FName", Paths. make(Person#FirstName)), QuerySelectColumns.path(Paths.make(Person#LastName)) }).orderBy(QuerySelectColumns.path(Paths.make(Person#LastName)))</pre>
	<i>alias</i> : String	An alias for the column name in the row query result. This parameter has no purpose for a column that you use to order the query results.
	<i>path</i> : PersistentPath	A path expression that references the column to include in the row query result or to order the rows in the result set.
dbFunctionWithAlias dbFunction		<p>Creates an SQL expression for ordering query results or to use as a column in a row query. For example, the following lines select columns for a row query and order the results. You use the same syntax for ordering the results of an entity query.</p> <pre>var query = Query.make(Address) var results = query.select({ QuerySelectColumns.dbFunction(DBFunction.Expr({ "length (", query.getColumnRef("Address.City"), ", " })) }).orderBy(QuerySelectColumns.dbFunction(DBFunction.Expr ({ "length(", query.getColumnRef("Address.City"), ", " })))</pre>
		<p>WARNING: Using dbFunction can cause the database to perform a whole-table scan. Ensure that use of these methods does not cause an unacceptable delay to the user interface.</p>
	<i>alias</i> : String	An alias for the column name in the row query result. This parameter has no purpose for an expression that you use to order the query results.
	<i>func</i> : DBFunction	A DBFunction expression that creates a column to include in the row query result or to order the rows in the result set.

Other methods to access columns in building a query include the following:

Method	Parameter	Description
getColumnRef		<p>Creates a reference to an entity field in terms of the database column that stores its values. Returns a ColumnRef object. This method does not support chaining. Use this method to create a column reference for a second property in a comparison predicate or for an argument to a DBFunction method.</p> <pre>var query = Query.make(Address) var rowResults = query.select({ QuerySelectColumns.pathWithAlias("Country", Paths.make(Address#Country)), QuerySelectColumns.dbFunctionWithAlias("LatestAddress", DBFunction.Max(Paths.make(Address#createTime))) })</pre>
	<i>propertyName</i> : String	A String of format " <i>Entity.Property</i> ".

Column specification in query results

To access a column in the results of an entity query, you use dot notation to access any property in the entire entity graph. For example, the following code prints the display name of a Person entity instance and the name of the user who created that Person instance. The default property is the display name, `DisplayName`.

```
var query = Query.make(Person)
var results = query.select()
for (person in results) {
    print(person.DisplayName + " " + person.CreateUser.Contact)
}
```

To access a column in the results of a row query, you use the `getColumn` method to access any column in the row. The results of a row query do not have a default property. For example, the following code is equivalent to the previous code for an entity query. The code creates a row query and prints the name of each person in the database and the name of the user who created that person.

```
var query = Query.make(Person)
var results = query.select({
    QuerySelectColumns.pathWithAlias("FName", Paths.make(Person#FirstName)),
    QuerySelectColumns.path(Paths.make(Person#LastName)),
    QuerySelectColumns.path(Paths.make(Person#CreateUser, User#Contact, Person#FirstName)),
    QuerySelectColumns.path(Paths.make(Person#CreateUser, User#Contact, Person#LastName))
})
for (person in results) {
    print(person.getColumn("FName") + " " + person.getColumn("Person.LastName") + " "
        + person.getColumn(2) + " " + person.getColumn(3))
}
```

The method that you use to retrieve the value of a column from a row in the result of a row query is the following. Use one of the parameters in the table. The return value of the method is an `Object` object.

Method	Parameter	Description
<code>getColumn</code>		Retrieves the value of a column from a row in the result of a row query.
	<code>alias : String</code>	An alias for the column name in the row query result. Use this parameter if you defined the column by using <code>pathWithAlias</code> or <code>dbFunctionWithAlias</code> .
	<code>propertyName : String</code>	A String of format <code>"Entity.Property"</code> . Use this parameter if you defined the column by using <code>path</code> or <code>dbFunction</code> .
	<code>position : int</code>	A zero-based int that is the numeric position of the column in the argument to the <code>select</code> method that created the results.

Predicate methods reference

The following table lists the types of comparisons and matches you can make with methods on the query object.

Predicate method	Parameter (Type)	Description
<code>and</code>	Gosu block that contains a list of predicate methods applied to columns in the query.	Checks whether a value satisfies a set of predicate methods, such as <code>compare</code> , <code>contains</code> , and <code>between</code> . All of the predicate methods must evaluate to true for the item that contains the value to be included in the result.
<code>between</code>	<ul style="list-style-type: none"> • Column name (String) • Start value (Object) • End value (Object) 	<p>Checks whether a value is between two values. This method supports String values, date values, and number values.</p> <pre>query.between(Activity#PublicID, "abc:01", "abc:99")</pre> <p>To specify an unbounded range on the lower or upper end, pass <code>null</code> as the first or second range argument but not both. You can pass <code>null</code> as a parameter regardless of null restrictions on the column.</p>

Predicate method	Parameter (Type)	Description
compare	<ul style="list-style-type: none"> • Column name (String) • Operation type (Relop) • Value (Object) 	<p>Compares a column to a value. For the operation type, pass one of the following values to represent the operation type:</p> <p>Equals Matches if the values are equal</p> <p>NotEquals Matches if the values are not equal</p> <p>LessThan Matches if the row's value for that column is less than the value passed to the compare method</p> <p>LessThanOrEqual Matches if the row's value for that column is less than or equal to the value passed to the compare method</p> <p>GreaterThan Matches if the row's value for that column is greater than the value passed to the compare method</p> <p>GreaterThanOrEqual Matches if the row's value for that column is greater than or equal to the value passed to the compare method</p> <p>Pass these values without quote symbols around them. These names are values in the Relop enumeration.</p> <p>For the value object, you can use numeric types, String types, ClaimCenter entities, keys, or typekeys. For String values, the comparison is case-sensitive.</p> <p>Example of a simple equality comparison:</p> <pre>query.compare(Activity#Priority, Equals, 5)</pre> <p>Example of a simple less than or equal to comparison:</p> <pre>query.compare(Activity#Priority, LessThanOrEqual, 5)</pre> <p>To compare the value to the value in another column, generate a column reference and pass that instead.</p> <p>You can use algebraic functions that evaluate to an expression that can be evaluated at run time to be the appropriate type. For example:</p> <pre>var prefix = "abc:" var recordNumber = "1234" query.compare(Activity#PublicID, Equals, prefix + recordNumber)</pre> <p>Or combine a column reference and algebraic functions:</p> <pre>query.compare(Activity#Priority, Equals, DBFunction.Expr({ query.getColumnRef("OldPriority"), "+", "10"}))</pre>
compareIgnoreCase	<ul style="list-style-type: none"> • Column name (String) • Operation type (Relop) • Value (Object) 	<p>Compares a character column to a character value while ignoring uppercase and lowercase variations. For example, if the following comparison succeeds:</p> <pre>query.compare("Name", Equals, "Acme")</pre>

Predicate method	Parameter (Type)	Description
		Both of the following comparisons also succeed: <pre>query.compareIgnoreCase("Name", Equals, Acme") query.compareIgnoreCase(Company#Name, Equals, "ACME")</pre>
compareIn	<ul style="list-style-type: none"> Column name (String) List of values that could match the database row for the column (Object[]) 	Compares the value for this column for each row to a list of non-null objects that you specify. If the column value for a row matches any of them, the query successfully matches that row. For example: <pre>query.compareIn(Activity#PublicID, {"default_data:1", default_data:3"})</pre>
compareNotIn	<ul style="list-style-type: none"> Column name (String) List of values that could match the database row for that column (Object[]) 	Compares the value for this column for each row to a list of non-null objects that you specify. If the column value for a row matches none of them, the query successfully matches that row. For example: <pre>query.compareNotIn(Activity#PublicID, {"default_data:1", default_data:3"})</pre>
contains	<ul style="list-style-type: none"> Column name (String) Contains value (String) Ignore case (Boolean) 	Checks whether the value in that column for each row contains a specific substring. For example, if the substring is "jo", it will match the value "anjoy" and "job" but not the values "yo" or "ji". If you pass true to the final argument, Gosu ignores case differences in its comparison. For example: <pre>query.contains(Person#FirstName, "jo", true /* ignore case */)</pre>
		Test the use of the contains method in a realistic environment. Using the contains method as the most restrictive predicate on a query causes a full-table scan in the database because the query cannot use an index.
WARNING:	For a query on a large table, using contains as the most restrictive predicate can cause an unacceptable delay to the user interface.	
or	Gosu block that contains a list of predicate methods applied to columns in the query.	Checks whether a value satisfies one or more predicate methods, such as compare, contains, and between. Only one of the predicate methods must evaluate to true for the item that contains the value to be included in the result.
startsWith	<ul style="list-style-type: none"> Column name (String) Substring value (String) Ignore case (Boolean) 	Checks whether the value in that column for each row starts with a specific substring. For example, if the substring is "jo", it will match the value "john" and "joke" but not the values "j" or "jar". If you pass true to the Boolean argument (the third argument), Gosu ignores case differences in its comparison. For example: <pre>query.startsWith(Person#FirstName, "jo", true /* ignore case */)</pre>
		Note: If you choose case-insensitive partial comparisons, Gosu generates an SQL function that depends on your ClaimCenter and database configuration to implement the comparison predicate. However, if the data model definition of the column specifies the supportsLinguisticSearch attribute set to true, Gosu uses the denormalized version of the column, instead.
IMPORTANT:	Test the use of the startsWith method in a realistic environment. Using the startsWith method as the	

Predicate method	Parameter (Type)	Description
		most restrictive predicate on a query can cause a delay on the user interface.
subselect	The arguments provide set inclusion and exclusion predicates.	Perform a join with another table and select a subset of the data by combining tables.
withinDistance	<ul style="list-style-type: none"> • Identifier for column of type SpatialPoint (IQueryablePropertyInfo) • Path to location or spatial property. Use only in complex entity queries, such as those containing a join. (String) • Center (SpatialPoint) • Distance (Number) • Unit of distance (UnitOfDistance) 	<p>Checks whether a location or spatial property on a column identifies a location that is within a given number of distance units of a center or reference location. For example:</p> <pre>addressQuery.withinDistance(Address.SPATIALPOINT_PROP.get(), SAN_MATEO, 10, UnitOfDistance.TC_MILE)</pre> <pre>addressTable.withinDistance(Address.SPATIALPOINT_PROP.get(), "Person.PrimaryAddress.SpatialPoint", SAN_MATEO, 10, UnitOfDistance.TC_MILE)</pre>

Predicate methods that support DBFunction arguments

The predicate methods that support a DBFunction object as the comparison value are:

- compare
- compareIgnoreCase
- between
- startsWith
- contains
- subselect

See also

- “Combining predicates with AND and OR logic ” on page 763
- “Comparing column values with each other” on page 760
- “Using set inclusion and exclusion predicates” on page 760

Aggregate functions in the query builder APIs

The following methods on the DBFunction class support aggregate queries in the query builder APIs.

Method	Description	Example
Avg	Returns the average of all values in a column	<pre>query.compare(DBFunction.Constant(44), GreaterThan. DBFunction.Avg(query.getColumnRef("B")))</pre>
Count	Returns the number of rows in a column	<pre>query.compare(DBFunction.Constant(44), GreaterThan, DBFunction.Count(query.getColumnRef("B")))</pre>
Min	Returns the minimum value of all values in a column	<pre>query.compare(DBFunction.Constant(44), GreaterThan, DBFunction.Min(query.getColumnRef("B")))</pre>
Max	Returns the maximum value of all values in a column	<pre>query.compare(DBFunction.Constant(44), LessThan, DBFunction.Max(query.getColumnRef("B")))</pre>

Method	Description	Example
Sum	Returns the sum of all values in a column	<pre>query.compare(DBFunction.Constant(44), GreaterThan, DBFunction.Sum(query.getColumnRef("B")))</pre>

Utility methods in the query builder APIs

The following methods on the DBFunction class provide useful functions in query builder code.

Method	Description	Example
Constant	Coerces a Gosu literal to an SQL literal in the generated SQL query that ClaimCenter submits to the database	<pre>query.compare(Address#City, Equals, DBFunction.Constant("Los Angeles"))</pre>
DateDiff	Returns the interval between two date and time columns	<pre>query.compare(DBFunction.DateDiff(DAYS, getColumnRef("AssignmentDate"), getColumnRef("EndDate")), LessThan, 15)</pre>
DatePart	Returns parts of a date and time column value, such as day or month	<pre>query.compare(DBFunction.DatePart(DAY_OF_MONTH, query.getColumnRef("AssignmentDate")), NotEquals, 15)</pre>
DateFromTimestamp	Returns the date portion of a timestamp	<pre>query.compare(DBFunction.DateFromTimestamp(query.getColumnRef("CreateTime")), Equals, gw.api.util.DateUtil. currentDate())</pre>
Distance	Returns the distance between the location of an entity instance and a reference location	<pre>QuerySelectColumns.dbFunctionWithAliasDistance", DBFunction.Distance(addressQuery, "Address.SpatialPoint"))</pre>
Expr	Returns a function defined by a list of column references and character sequences	

See also

- “Comparing the interval between two date and time fields” on page 756
- “Comparing parts of a date and time field” on page 757
- “Comparing the date part of a date and time field” on page 757
- “Comparing a column value with a literal value” on page 761
- “Creating and using an SQL database function” on page 785

Database connection pool

Connections to the database connection pool can be automatic or, to improve performance, reserved.

See also

- “<dbcp-connection-pool> database configuration element” in the *System Administration Guide*

Reserving a single database connection

Whenever a database query occurs, a connection to the database is retrieved from the database connection pool. When the query is finished, the connection is released and returned to the pool.

This behavior works well in most situations. However, special cases can exist where retrieving and releasing a connection for each query might not be appropriate. For example, an operation that performs a large number of database queries in succession would experience excessive overhead if it retrieved and released a connection for each query. To avoid this situation, a single database connection can be retrieved from the pool and then reused for each query. When the series of queries has finished, the connection is released back to the pool.

A database connection can be retrieved from the pool and reused by multiple database queries by calling the `executeTransactionsWithReservedConnection` method of the `ConnectionUtil` class. The `ConnectionUtil` class is located in the `gw.transaction` package.

```
executeTransactionsWithReservedConnection(Runnable runnable)
```

The `runnable` argument is a `Runnable` object that implements a `run` method that will use the single database connection for its queries. The method has no return value.

The `executeTransactionsWithReservedConnection` method retrieves the database connection prior to executing the argument's `run` method. It releases the connection when the `run` method has finished. The `run` method can retrieve the reserved connection by calling the `getActiveConnection` method of the `ConnectionHandlerFactory` class.

```
Runnable block = new Runnable() {
    @Override
    public void run() {
        /* Retrieve the reserved database connection */
        ConnectionHandler handler = ConnectionHandlerFactory.getActiveConnection();

        /* ... Perform desired operations ... */
        /* When the method returns or throws an exception, the connection is automatically
         * released back to the pool.
        */
    }
}
```

```
/* Reserve a database connection from the pool and run the block object */
executeTransactionsWithReservedConnection(block);
```

A web service can reserve and release a single database connection for its own use by specifying the `@WsiReduceDBConnections` annotation.

See also

- For information about the web service `@WsiReduceDBConnections` annotation, see “Web service publishing annotation reference” on page 73.

Reference specifications

Integration mappings map Guidewire InsurancePlatform Integration Views to a serialized form that is defined by a JSON schema. The integration mapping specification describes mapping files, mappers, mapping imports, filters, and the `JsonMapper` and `TransformResult` classes.

The JSON schema documents that ClaimCenter uses support a subset of the JSON Schema, Draft 4 syntax. Additionally, ClaimCenter defines other components that support the definition of stable, versioned contracts for data that you send to external systems. These components include:

- Parser and serializer of JSON data
- Code generator of statically typed wrapper classes
- Integration mapping files
- Swagger schemas

Integration mapping specification

The reference information on integration mappings covers mapping files, mappers, mapping imports, filters, and the `JsonMapper` and `TransformResult` classes. The final part of the reference information is a set of integration mapping examples.

What are integration mappings?

Integration mappings are one part of the integration frameworks defined in Guidewire InsuranceSuite. These mappings are part of Integration Views. Integration mappings permit declarative mapping files that describe how to map a given root object into JSON or XML data that conforms to a specified JSON schema document. The declarative mapping files can be of an entity type or a standard Java or Gosu object.

Integration View components

An Integration View consists of three parts:

- A JSON schema that defines the structure of the data
- An integration mapping that describes how to transform a given source object into a JSON or XML document that conforms to the schema
- Optional GraphQL filters that whitelist the fields to materialize and serialize

Any given integration mapping targets a single JSON Schema, but there might be more than one integration mapping that targets a given JSON schema.

Integration mapping files

Integration mapping files contain one or more integration mappers. Each mapper describes how to transform a specific root object type into a specific JSON schema definition.

You can group an arbitrary number of integration mappers together into a single integration mapping. By grouping the mappers, you can allow them to easily reference each other and to be versioned and extended as a unit rather than just individually.

File names

Write integration mapping files in JSON. Place these files in a subdirectory of the `config/integration/mappings` directory. Any subdirectories of `/mappings` become parts of the fully-qualified name of the mapping file. This behavior is similar to the Java package structure.

Name the files themselves with the format, `<name>-<version>.mapping.json`:

- The <name> portion cannot contain the hyphen character or any other character that would render the file name invalid on some file systems.
- The <version> portion consists of one or more sequences of digits. Separate these sequences of digits with the period character. Optionally follow each sequence of digits with a hyphen and an arbitrary string.

Examples:

- config/integration/mappings/gw/pl/admin/user-1.0.mapping.json defines the mapping named gw.pl.admin.user-1.0.
- config/integration/mappings/gw/pc/productmodel/productmodel-10.0.1-alpha.mapping.json defines the mapping named gw.pc.productmodel.productmodel-10.0.1-alpha.

See also

- “Integration mapping file specification” on page 824
- “Integration mapping file combination” on page 826

Integration mapping file specification

General information

Requiredness

Mapper properties designated as required are not always necessary in every mapping file. You must consider such properties in light of the entire combined mapping. You need not respecify a required property if a combined mapping file already defines it.

For example, suppose an extension mapping file, address_ext-1.0, combines a base mapping file, address-1.0. Further suppose that the base mapping file defines an Address mapper, the extension mapping file can also define the Address mapper. However, the extension mapping file need not redefine the schemaDefinition or root properties even though these properties are required. The Address mapper in the extension mapping file inherits those property values from the Address mapper in the base mapping file.

Combination styles

The combination of mapping files is roughly equivalent to an ordered, textual merge of the files. However, there are some nuances around how the merge works. Depending on the object and property, the following styles are possible:

merged

The referenced subobjects are merged together based on the rules for combining the particular object.

merged by key

The merged by key combination style involves matching up objects based on the keys in a map. For example, elements under the properties property with the same key are merged.

first non-null

Given N objects to be merged together, the mapping takes the first non-null value for the given property.

not inherited

The property is explicitly not inherited across files. This style is generally for root-level defaults that are propagated within only a given file.

Root object

Property	Type	Required	Format	Behavior	Combination Style
schemaName	string	Yes	The fully-qualified name of the schema targeted by this set of mappings	N/A	First non-null

Property	Type	Required	Format	Behavior	Combination Style
combine	string[]	No	Array of fully-qualified integration mapping names	Specifies the integration mappings to combine with this file	Not inherited
import	map<string, string>	No	Keys are alias names, values are fully-qualified integration mapping names	Defines aliases for other integration mapping files to which this mapping can refer	Merged by key
mappers	map<string, Mapper Object>	No	Keys are the names of the mappers	The integration mappers that this mapping defines	Merged by key

Mapper object

Property	Type	Required	Format	Behavior	Combination Style
schemaDefinition	string	Yes	Name of a schema definition that exists in the schema defined output by schemaName	The name of the schema definition for which this mapper produces	First non-null
root	string	Yes	A type name	The root object type that this mapper takes as input. The relative name of the type will be used as the symbol exposed to the path and predicate properties on mapper properties.	First non-null
properties	map<string, Property Object>	No	Keys are the names of the properties	The set of properties for this mapper. In general, these properties are expected to match exactly the set of properties on the targeted schema definition.	Merged by key

Property object

Property	Type	Required	Format	Behavior	Combination Style
path	string	Yes	Gosu expression	<p>The Gosu expression to evaluate in order to produce the value of the associated schema property. This expression has a single symbol available, whose name is the relative name of the root type and whose type is exactly the root type. The expected return type of this expression depends upon the schema property this mapping property targets:</p> <ul style="list-style-type: none"> • If the schema property is a simple scalar, this expression will return an object that matches the schema property type. The return object is one of the Java inputs for the schema property type. • If the schema property is an array of scalars, this expression will return an Iterable object or an array whose elements match the schema property type. • If the schema property is an object property, this expression will return an object that is assignable to the root type of the referenced mapper. • If the schema property is an object array property, this expression will return an Iterable object or an array whose elements are assignable to the root type of the referenced mapper. 	First non-null

Property	Type	Required	Format	Behavior	Combination Style
mapper	string	No	Either #/ mappers/ <name> or <alias>#/br/>mappers/ <name>	References the mapper to use to produce output if this schema property corresponds to a JSON object or array of JSON objects. This property is required if the referenced schema property is an object or object array property, otherwise it is not allowed.	First non-null
predicate	string	No	Gosu expression	A predicate expression must return a Boolean object value or a boolean primitive value. If the expression is specified and returns false, the path expression is never evaluated. In addition, the path property is treated as if it had evaluated to a null value. The predicate property can be used to guard execution of the path expression. This guarding occurs in cases where the path expression is relevant for only a particular subtype of the root and contains a downcast.	First non-null

See also

- “Integration mapping file combination” on page 826

Integration mapping file combination

The combination process for integration mapping files and the target use cases for it are similar to the process and use cases for JSON schemas and Swagger schemas. Most of the “JSON schema file combination” on page 844 reference applies to integration mappings as well.

Integration mapping combination specifics

Integration mapping files are simple compared to JSON schemas and Swagger schemas. Hence, the combination process for mapping files is just as simple as the process is for schemas. Exact details for how to combine specific elements are in the “Integration mapping file specification” on page 824. To validate combined mappings, you follow a philosophy similar to the philosophy you would follow for validating the structure of JSON schema and Swagger schema combinations. If a mapping named `Ext` combines with a mapping named `Base`, the following rules would apply:

- If `Ext` changes the root `schemaName` from `BaseJson` to `ExtJson`, the schema definitions in `ExtJson` that the mapper uses must be compatible from a structural standpoint with the schema definitions in `BaseJson`.
- If `Ext` changes the root type of a mapper defined in `Base`, the root type defined in `Ext` must be contravariant with respect to the type defined in `Base`.
- If `Ext` changes a path expression that is defined in `Base`, the new path expression must have a return value that is covariant with respect to the expression defined in `Base`.

The net intent for these rules is that you be able to substitute references to mappers in `Ext` for references to mappers defined in `Base`. That is, if code refers to the `Address` mapper from the `Base` mapping, you can change that code to refer to the `Address` mapper from the `Ext` mapping.

In this case, the code will continue to execute at runtime. Moreover, the code will execute because the `Ext` version of the mapper accepts either the input type or a more generic type. In addition, the code will produce output that is a superset of the `Base` output. This relationship between outputs exists because the schema definitions in `Ext` are structurally compatible the definitions in `Base`.

Schema overrides

You can change the effect of downstream schemas to which the schema `Base` refers by modifying the mapping `Ext`. You effect this change by changing the root `schemaName` property on the mapping. ClaimCenter resolves mapper schema references with respect to the root `schemaName` property. Consequently, changing the root `schemaName` property will cause all mapper schema definitions to resolve in terms of the new schema. The common extension pattern looks like this:

1. A first schema, `base_schema-1.0`, is the starting point.
2. A first mapping, `base_mapping-1.0`, refers to `base_schema-1.0` in the mapping `schemaName` attribute.
3. A second schema, `ext_schema-1.0`, combines `base_schema-1.0` and adds new definitions and properties.
4. A second mapping, `ext_mapping-1.0`, combines `base_mapping-1.0` and uses `ext_schema-1.0` as a value for the `schemaName` attribute.

The mappers that the second mapping, `ext_mapping-1.0`, inherits from the first mapping, `base_mapping-1.0`, will be pointing to the schema definitions in `ext_schema-1.0`. The second schema, `ext_schema-1.0`, will in turn inherit the properties and definitions from `base_schema-1.0`. The second schema will also add new properties and definitions not in the first schema.

See also

- “JSON schema file specification” on page 836
- “Integration mapping file specification” on page 824

Integration mapping file imports

Overview

Integration mapping files define an `import` reference in addition to a `combine` composition. The `import` reference for mapping files is analogous to the `import` reference for JSON schema files.

In addition, the differences between the `import` reference and `combine` composition are roughly analogous. On the one hand, the `combine` composition will stitch together all mappers with the same name. On the other hand, the `import` reference will keep everything in a separate namespace.

Choose between the composition and the reference based on whether you want the combination behavior or the import behavior. The combination behavior is ideal when logically extending another mapping file. The import behavior is ideal when reusing a standard mapping or mapper for a shared schema.

Import syntax

Integration mapping imports look similar to JSON schema imports, but they use the `import` property name instead of `x-gw-import`. References to imported mappers have a prefix with the alias name as in the following example:

```
{  
  "schemaName": "gw.px.ete.contact-1.0",  
  "import" : {  
    "address" : "gw.px.ete.address-1.0"  
  },  
  "mappers": {  
    "Contact" : {  
      "schemaDefinition" : "Contact",  
      "root" : "entity.Contact",  
      "properties" : {  
        "EmailAddress1" : {  
          "path" : "Contact.EmailAddress1"  
        },  
        "HomePhone" : {  
          "path" : "Contact.HomePhone"  
        },  
        "HomePhoneCountry" : {  
          "path" : "Contact.HomePhoneCountry"  
        },  
        "AllAddresses" : {  
          "path" : "Contact.AllAddresses",  
          "mapper" : "address#/mappers/Address"  
        },  
        "PrimaryAddress" : {  
          "path" : "Contact.PrimaryAddress",  
          "mapper" : "address#/mappers/Address"  
        }  
      }  
    }  
  }  
}
```

Import overrides

Just like with JSON schema imports, you can override mapping file imports in a combined mapping by re-declaring the imported alias with a different fully-qualified mapping name:

```
{
  "schemaName" : "customer.contact-1.0",
  "combine" : ["gw.px.ete.contact-1.0"],
  "import" : {
    "address" : "customer.address-1.0"
  }
}
```

Overriding a mapping file import will cause all inherited mapper references to the `address` alias to reference mappers from the `customer.address-1.0` mapping rather than the `gw.px.ete.address-1.0` mapping.

Integration mappers

Integration mappers

Each integration mapping file can have any number of integration mappers defined in it. Each integration mapper defines a single entry point into the mapping transformation. An integration mapper takes a single input object and transforms it into JSON or XML syntax that matches an output schema. Each integration mapper contains mapping properties that correspond to each property in the output schema. Each integration mapper defines the following:

A root type

Both the expected runtime input to this mapper and the base type of the symbol available to path and predicate expressions.

A schema definition

The JSON Schema definition whose output this mapper is producing. The schema definition is resolved in the context of the root schema name for the mapping file.

A set of properties

Definitions for obtaining data for each property in a referenced schema definition. A mapper is not technically required to specify every property on the referenced schema definition. However, a warning will be issued if there are properties on the schema definition that are not being mapped.

Note that the mapper name is not required to match the name of the schema definition it targets. In fact, you may legally have multiple mappers that target the same schema defined within the same file. For example, the mappers might take different root object types but produce output that conforms to the same schema.

Mapping properties

The properties on an integration mapper must correspond to properties on the associated JSON Schema definition. Moreover, the names of the keys in the mapper properties are assumed to be the names of the JSON schema definition properties. Effectively, the integration mapper is defining the output directly in terms of the schema. The integration mapper is then specifying how to get the value for each property that can appear in the output.

path

Each mapping property must specify a path expression. This expression is a Gosu expression that evaluates to the value that must be used as the output of that schema property. The path expression has a single symbol available to it. The single symbol has a type equal to the root type of the mapper. The single symbol also has a name equal to the relative name of the type. For example, suppose that the root type for a mapper is `entity.Contact`. The path expressions on the mapper properties will have a single symbol named `Contact` with a type of `entity.Contact`. The path expression must evaluate to something appropriate for the referenced schema property:

- If the schema property is a scalar property, the path expression must evaluate to a Java Input Type listed in “JSON data types and formats” on page 847. The evaluation is based on the `type/format/x-gw-type` of the schema property. For example, if the schema type is `string` and the format is `date-time`, the path expression must evaluate to something assignable to `java.util.Date`.

- If the schema property is an array of scalars, the path expression must evaluate to an `Iterable` or array. The elements of the `Iterable` or array can be inputs to the data conversion for the type of the schema property items.
- If the schema property is an object property, then the `mapper` property must be specified. In addition, the return type of the path expression must be assignable to the root type of the referenced mapper. For example, if the referenced mapper is `#/mappers/Address` and the `Address` mapper has a root type of `entity.Address`, the path expression must evaluate to an `entity.Address` or a subtype.
- If the schema property is an object array property, then the `mapper` property must be specified. In addition, the path expression must evaluate to an `Iterable` or array whose elements can be assigned to the root type of the referenced mapper.

There is no automatic coercion performed for path expressions. Moreover, if a schema property is of type `string` with no format or is `x-gw-type`, the Gosu path expression must evaluate to a `java.lang.String`. This evaluation must be with no implicit invocation of `toString` or any similar method.

The path expression can be an arbitrary Gosu expression. The expression is not required to be a simple property path. The expression could be a method call, a static method call, or another complex Gosu expression.

predicate

The `predicate` property is an optional `boolean` or `Boolean` predicate that will be evaluated before the path expression is evaluated. The `predicate` property can also return a `java.lang.Boolean` type. However, the property cannot evaluate to `null` at runtime. If the `predicate` expression evaluates to `false`, the path expression will never be evaluated and the property will be treated as having a `null` value. The `predicate` expression can be used to guard the evaluation of the path expression. This guarding can be useful if the output schema flattens subtype columns onto a single object. For example, suppose that the output schema for `Contact` defines properties `firstName` and `lastName`. In this case, the `predicate` expression might be `Contact typeis Person`, and the path expression could be `(Contact as Person).FirstName`. Using the `predicate` expression in this way allows the path expression to downcast directly instead of using a more awkward ternary expression.

mapper

The `mapper` property defines the mapper to use when the `schema` property is an object or array of objects. Mapper references use a similar syntax as schema references. They must be of the form `#/mappers/<name>` when the referenced mapper is defined in the same mapping. They must be of the form `<alias>#/mappers/<name>` when the mapper is imported as `<alias>`. The referenced mapper must have a `schemaDefinition` property that matches the `$ref` on the `schema` property. For example, suppose that the `Contact` JSON schema definition has a `primaryAddress` schema property with a `$ref` of `#/definitions/Address`. In this case, the `primaryAddress` property on the `Contact` integration mapper must have a `mapper` reference where the `schemaDefinition` is `Address`. See “Integration mapping examples” on page 833 for an example of how the structure of integration mapping properties and mapper references mimic the structure of JSON schema properties and definition references.

- “Integration mapping file specification” on page 824
- “JSON data types and formats” on page 847
- “Integration mapping examples” on page 833

Integration View filters

Overview

It is often the case that two different external systems need substantially similar views of the same data but not exactly the same view. This case could arise when two different downstream systems both need policy information but might also need different levels of detail about coverage options or transaction data. The case could also arise when two different client systems want to display a different subset of the data to their end users. Developers putting together an integration message or building a REST API have to decide if they want to create a single schema that represents the superset of data that all such systems might want. In the alternative, developers have to decide if they must create

separate, more targeted schemas that only contain exactly what a particular system or use case requires. Having a single, shared schema reduces implementation and maintenance costs but comes at the cost of fetching, processing, and serializing out data that a given client will simply ignore.

The Integration Views feature attempts to meet these competing needs by allowing you to apply a filter to a schema and mapping file. The filter serves as a whitelist of the properties that must be included for a given invocation of a mapping file. Properties that are not included as part of a given filter never have their associated path or predicate expressions executed. These properties will not end up in the resulting `TransformResult` object at all. Not including unnecessary properties saves the cost of fetching, processing, and serializing the associated data.

Filters can also be used to create a stable view on top of a given schema. By whitelisting the properties desired, newly added properties will never show up unless the filter is explicitly changed.

Syntax

Filters make use of a simplified version of GraphQL syntax to describe the data to be fetched. See <https://graphql.org> and <http://facebook.github.io/graphql/October2016> for more details about the specification. The queries all have an implicit entry point based on their invocation and do not support arguments or mutations. They simply look like a JSON object with keys but no values.

For example, suppose we have a JSON schema that minimally defines activities and notes:

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "definitions": {
    "Activity": {
      "type": "object",
      "properties": {
        "assignedUser": {
          "type": "string"
        },
        "assignmentDate": {
          "type": "string",
          "format": "date-time"
        },
        "assignmentStatus": {
          "type": "string",
          "x-gw-type": "typekey.AssignmentStatus"
        },
        "createUser": {
          "type": "string"
        },
        "notes": {
          "type": "array",
          "items": {
            "$ref": "note#/definitions/Note"
          }
        },
        "subject": {
          "type": "string"
        }
      }
    },
    "Note": {
      "type": "object",
      "properties": {
        "created": {
          "type": "string",
          "format": "date"
        },
        "subject": {
          "type": "string"
        },
        "body": {
          "type": "string"
        },
        "topic": {
          "type": "string",
          "x-gw-type": "typekey.NoteTopicType"
        }
      }
    }
  }
}
```

The following GraphQL query will filter the resulting data, retrieving just the subject and notes for the `Activity` but ignoring the other properties. The query retrieves just the subject, body, and topic from the `Note`.

```
{  
  subject  
  notes {  
    subject  
    body  
    topic  
  }  
}
```

Objects and object array references

As illustrated in the previous example, referencing an object or object array in a GraphQL query requires that you specify the fields on the subobject you would like to retrieve.

Fragments

Fragments are a GraphQL feature that allows you to query fragments that can be applied to a given object.

Fragments are useful in cases where the same subobject type appears within multiple places in the data you want to fetch. For example, a `Contact` may have both `primaryAddress` and `allAddresses` properties that map to the same `Address` type. Contacts may also appear in multiple places within a given schema graph. Fragments give you a way to define the set of properties you want from a `Contact` or `Address` once, as a fragment. You can then reference that fragment in each context that you want the same set of properties. The previous example with activities and notes can be rewritten such that the note fields are defined as a fragment:

```
{  
  subject  
  notes {  
    ...noteParts  
  }  
}  
  
fragment noteParts on Note {  
  subject  
  body  
  topic  
}
```

Differences from GraphQL proper

Our filter syntax leverages the basics of GraphQL for the syntax of basic field selection, nested objects, arrays, and fragments. However, the syntax differs from GraphQL proper in some important ways:

- No explicit query operation exists. The query operation is always implied.
- No mutations are supported.
- None of our fields accept arguments.
- Fields cannot be aliased to change the name of the result property.
- Our GraphQL syntax allows for using the original unicode name of a schema property rather than restricting everything to a limited ASCII subset.

Storage

Filter text is stored in files under `config/integration/filters` with names of the form `<name>-<version>.gql`. As with JSON schema files, integration mapping files, and Swagger schemas; the fully-qualified name of the filter is formed first based on the package of the file indicated by the file subfolder. Then, the fully-qualified filter name relies upon the name and version of the file. For example, if the above query was in the file `config/integration/filters/gw/pl/activities/activity_basics-1.0.gql`, the fully-qualified name of the filter would be `gw.pl.activities.activity_basics-1.0`.

Usage

In order to use a given filter, you specify it on the `JsonMappingOptions` that you can optionally pass to a `JsonMapper` during invocation:

```
var activity : Activity
var mapper = JsonConfigAccess.getMapper("gw.pl.activities.activity-1.0", "Activity")
var transformResult = mapper.transformObject(activity, new
JsonMappingOptions().withFilter("gw.pl.activities.activity_basics-1.0"))
```

JsonMapper and TransformResult

JsonMapper

The `JsonMapper` interface provides runtime access to execute a mapper. The general way to obtain a `JsonMapper` instance involves two steps. First, use the `JsonConfigAccess` class to get a handle to the `JsonMapping`. Second, looking up the mapper by name. Both of the following approaches are equivalent:

```
var mapping = JsonConfigAccess.getMapping("gw.pl.contact-1.0")
var mapper = mapping.Mappers.get("Contact")

// Or you can use the helper method on JsonConfigAccess
var mapper = JsonConfigAccess.getMapper("gw.pl.contact-1.0", "Contact")
```

Once you have access to a `JsonMapper` instance, you can then invoke one of the `transformObject` or `transformObjects` method variants to obtain a `TransformResult` object:

```
var contact : Contact // Presumably the Contact is coming from somewhere:
// from the event message rule context, from a query, etc.
var mapper = JsonConfigAccess.getMapper("gw.pl.contact-1.0", "Contact")
var transformResult = mapper.transformObject(contact)
```

JsonMappingOptions

The `transformObject` methods can optionally take a `JsonMappingOptions` object that can change some aspects of how the transform is performed. The `JsonMappingOptions` object is the mechanism for applying a GraphQL filter to the mapping. Take the following code for example:

```
var contact : Contact
var mapper = JsonConfigAccess.getMapper("gw.pl.contact-1.0", "Contact")
var transformResult = mapper.transformObject(contact,
    new JsonMappingOptions().withFilter
        ("gw.pl.contact.contact_summary-1.0"))
```

You can also use the `JsonMappingOptions` object to disable the automatic tracking that prevents infinite recursion during mapping execution. You can also use the object to allow the transform to complete even if there are exceptions thrown during the mapping. An example of this use case is if you are serializing the output even in the presence of errors for debugging purposes.

TransformResult

The `TransformResult` object represents the results of a completed transform call. This object contains the results of executing each of the path expressions against a supplied root object. The `TransformResult` object is an intermediate representation of the data that can then be serialized out to either JSON or XML. For more details of how ClaimCenter translates JSON and JSON Schema into XML and XSD, see the *REST API Framework*. The `TransformResult` object is similar to a `JsonObject` in that it is mainly a map of property names to values. However, the object differs from a `JsonObject` in that each property value embeds information about the JSON schema property that it represents. Serializing out a `JsonObject` requires passing the JSON schema definition as an argument during serialization. By contract, the `TransformResult` object has schema information already embedded in it and always serializes based on that schema. So you can just invoke methods like `toPrettyJsonString` or `toXmlString` directly:

```
var contact : Contact
var mapper = JsonConfigAccess.getMapper("gw.pl.contact-1.0", "Contact")
var transformResult = mapper.transformObject(contact)
```

```
var jsonString = transformResult.toJsonString()
var xmlString = transformResult.toXmlString()
```

The `TransformResult` object also exposes some methods to allow traversing the tree of values explicitly. You could potentially use these methods to build custom serialization formats for the data.

Serialization options

The various serialization methods on `TransformResult` objects can take an optional `JsonSerializationOptions` object. This object controls aspects of how serialization works. By default, serializing a `TransformResult` object will ignore `null` properties, `null` list items, and empty arrays. Note that this last default is different from default `JsonObject` serialization. The reason is that entity arrays are never `null`. Mappings will often be used with entities. You can use the `JsonSerializationOptions` object to change that serialization behavior. However, note that including `null` properties or elements will require that the associated JSON schema properties or items be marked as `x-gw-nullable`.

The `JsonSerializationOptions` object can also be used to include some special properties on the root object when the data is serialized. Such properties can include the fully-qualified name of the schema definition, the correlation ID of the current request, or the timestamp at which the transform was executed. These options are only valid for serialization to JSON. See the javadoc on `JsonSerializationOptions` for more details.

Usage in REST

In REST API implementations, a `TransformResult` object can be returned directly by a handler method or embedded as the entity in a `Response` object. There is no need to explicitly call any of the serialization methods like `toJsonString`. The REST API Framework itself will serialize the `TransformResult` object appropriately based on the negotiated content type of the response.

Integration mapping examples

How JSON schemas relate to integration mappings is easiest to see if you view them next to one another. In the following example are a simple schema named `contact-1.0` and the integration mapping that corresponds to it:

<pre>{ "\$schema": "http://json-schema.org/draft-04/schema#", "definitions": { "Address": { "type": "object", "properties": { "addressLine1": { "type": "string" }, "city": { "type": "string" }, "state": { "type": "string", "x-gw-type": "typekey.State" } } }, "Contact": { "type": "object", "properties": { "allAddresses": { "type": "array", "items": { "\$ref": "#/definitions/Address" } }, "displayName": { "type": "string" }, "firstName": { "type": "string" }, "lastName": { "type": "string" }, "primaryAddress": { "\$ref": "#/definitions/Address" }, "secondaryAddress": { "\$ref": "#/definitions/Address" } } } } }</pre>	<pre>{ "schemaName": "contact-1.0", "mappers": [{ "Address": { "schemaDefinition": "Address", "root": "entity.Address", "properties": { "addressLine1": { "path": "Address.addressLine1" }, "city": { "path": "Address.City" }, "state": { "path": "Address.State" } } }, "Contact": { "schemaDefinition": "Contact", "root": "entity.Contact", "properties": { "allAddresses": { "path": "Contact.AllAddresses", "mapper": "#/mappers/Address" }, "displayName": { "path": "Contact.DisplayName" }, "firstName": { "path": "(Contact as Person).FirstName", "predicate": "Contact typeis Person" }, "lastName": { "path": "(Contact as Person).LastName", "predicate": "Contact typeis Person" }, "primaryAddress": { "path": "Contact.PrimaryAddress", "mapper": "#/mappers/Address" } } } }] }</pre>
--	---

The pieces match up as follows:

- The `schemaName` property of the mapping file references the fully-qualified name of the JSON schema file. The `schemaDefinition` property within the `Address` and `Contact` mappers references definitions within the schema.
 - The properties on each mapper exactly match the set of schema properties on the associated schema definitions.
 - The `Address` mapper has a root type of `entity.Address`. Consequently, all path and predicate expressions on properties for the `Address` mapper have the symbol `Address` available with a type of `entity.Address`. Similarly, the `Contact` mapping properties all have the `Contact` symbol available to their Gosu expressions.
 - For scalar properties, the path expressions all evaluate to an object type that is compatible with the data type implied by the `type/format/x-gw-type` on the schema property. The ramifications include the following:
 - `Address.addressLine1` is a `String`
 - `Address.city` is a `String`
 - `Address.state` is a `typekey.State`
 - `Contact.displayName` is a `String`
 - `Contact.firstName` is a `String`
 - `Contact.lastName` is a `String`
 - `Contact.subtype` is a `typekey.Contact`
 - For object or object array properties, the path expression returns something that matches the root type of the referenced mapper, which means the following:
 - `Contact.allAddresses` returns an `entity.Address[]`
 - `Contact.primaryAddress` returns an `entity.Address`
 - The `Contact.firstName` and `Contact.lastName` mapping properties use a predicate so that the path expression will only execute if the `Contact` is actually a `Person`. This arrangement allows the path expression to be cleaner. The arrangement also avoids awkward ternary expressions.

Guidewire JSON schema support specification

Guidewire InsuranceSuite has added support for JSON Schema documents that use a subset of JSON Schema, Draft 4 syntax. InsuranceSuite has also added additional components that work with JSON data and schema files. These components include:

- Parser and serializer of JSON data based on a schema definition using the `JsonObject` class.
- Code generator of statically-typed wrapper classes based on JSON schema object definitions. The generator layers statically-typed getters and setters on top of a `JsonObject`.
- Integration mapping files that can declaratively produce data that conforms to a schema.
- Swagger schemas for defining REST APIs. These schemas can refer to the JSON schema types for requests and responses.

InsuranceSuite always gives JSON schema files an explicit, fully-qualified name, which always includes a version number.

These facilities allow you to define stable, versioned contracts for data that you send to external systems. You can use these contracts for data that you transit either through event messages or from REST APIs. You can also use the contracts for data that you input to the system through REST APIs or other custom input paths.

JSON schema files

JSON schema files define the structure of a given set of JSON objects. Guidewire InsuranceSuite JSON Schema support covers a subset of the fourth draft of the JSON Schema standard.

JSON schema files have three uses. JSON schema files can be output targets for integration mappings. In addition, they can serve as input or output types for Swagger schemas. You can also use JSON schema files standalone to parse or serialize JSON objects.

Note: Only Swagger schemas define REST APIs; JSON schemas cannot. Also, a Swagger schema must reference a JSON schema file if any of the operations in the Swagger schema have a JSON input or output.

File names

You write JSON Schema files in the JSON format. Place the files in a subdirectory of `config/integration/schemas` in Guidewire Studio.

Name schema files with the format, `<name>-<version>.schema.json`. When doing so, note the following:

- The `<name>` portion cannot contain the hyphen character or any other character that would render the portion an invalid file name. The portion also does not include the version number.
- The `<version>` portion consists of one or more sequences of digits. Separate these sequences of digits with the period character. Optionally follow each sequence of digits with a hyphen and an arbitrary string.

Examples:

- `config/integration/schemas/gw/pl/admin/user-1.0.schema.json` defines the schema named `gw.pl.admin.user-1.0`.
- `config/integration/schemas/gw/pc/productmodel/productmodel-10.0.1-alpha.schema.json` defines the schema named `gw.pc.productmodel.productmodel-10.0.1-alpha`.

Note: Distinguish schema file names from fully-qualified schema names. Format a schema file name in the manner this topic describes. Use a fully-qualified schema name to reference a schema file in a mapping file, Swagger file, import reference, or combine composition. Such a schema name includes any subdirectories of `config/integration/schemas`. In this way, the schema name has a function similar to a Java package.

See also

- “JSON schema file specification” on page 836
- “JSON schema file combination” on page 844
- <https://tools.ietf.org/html/draft-zyp-json-schema-04>
- <https://tools.ietf.org/html/draft-fge-json-schema-validation-00>

JSON schema file specification

General information

Relation to JSON Schema Draft 4

JSON Schema support within Guidewire InsuranceSuite is based on JSON Schema Draft 4. However, the support is for a very limited subset of Draft 4. Guidewire must allow for consistency and simplicity across use cases. For example, this consistency and simplicity must be available for integration mappings that target a schema and for wrapper class generation. To facilitate consistency and simplicity, Guidewire imposes significant constraints on the shape of the schemas you can use. Among these constraints are the following:

- No support for `allOf`, `anyOf`, or `oneOf` keywords
- No support for `patternProperties` keyword
- No support for nested object definitions. You must explicitly name all object definitions. Afterwards, you can reference these top-level definitions.
- No support for named definitions that are not object types
- No support for items that have an `array` type. For example, you cannot use nested arrays.
- No support for heterogeneous items within a property

Note: Some of these restrictions match restrictions that the Guidewire flavor of Swagger 2.0 imposes. In addition, the Guidewire flavors of JSON Schema Draft 4 and Swagger 2.0 have many custom features.

Requiredness

Properties designated as required are not always necessary in every schema document. You must consider such properties in light of the entire combined schema.

For example, properties generally must specify one of either a `type` or `$ref` property. However, a special case arises if a schema is merely overriding the description on a property that the schema inherits from a combined schema. In this case, the extension schema does not legally have to specify either the `type` or `$ref` property.

Combination styles

The combination of schema files is roughly equivalent to an ordered, textual merge of the files. However, there are some nuances around how the merge works. Depending on the object and property, the following styles are possible:

merged

The referenced subobjects are merged together based on the rules for combining the particular object.

merged by key

The merged by key combination style involves matching up objects based on the keys in a map. For example, elements under the `properties` property with the same key are merged.

merge extensions

Extensions objects are merged by taking the first defined value for each key even if that defined value is explicitly `null`.

merge names

Required property names combine together to form a single list. A schema that combines cannot eliminate the required nature of a property from a definition that the schema is inheriting.

first non-null

Given N objects to be merged together, the mapping takes the first non-null value for the given property.

not inherited

The property is explicitly not inherited across files. This style is generally for root-level defaults that are propagated within only a given file.

n/a

The containing object or property is always treated as an atomic unit and never merged. For example, the `name` property on an `Items` XML Object is never merged. The mapping treats the two properties as a single unit that can be replaced wholesale such that the properties on the `name` object are never combined.

Documentation only properties

Many properties are listed as "Documentation only" for their behavior. Guidewire InsuranceSuite does not use the value of such properties to determine runtime behavior. However, InsuranceSuite does include the properties when it publishes a JSON schema for external consumption. Tools that consume a JSON schema or Swagger schema that contains these properties can use them.

Root object

Property	Type	Required	Format	Behavior	Combination Style
<code>\$schema</code>	string	Yes	<code>http://json-schema.org/draft-04/schema#</code>	Defines the JSON Schema SDL version	not inherited
<code>title</code>	string	No		Documentation only	first non-null
<code>description</code>	string	No		Documentation only	first non-null
<code>x-gw-combine</code>	string[]	No	Array of fully-qualified JSON schema names	Specifies the schemas to combine with this file	not inherited
<code>x-gw-import</code>	map<string, string>	No	Keys are alias names. Values are fully-qualified JSON schema names.	Defines aliases for other schema files to which this schema file can refer	merged by key
<code>x-gw-xml</code>	<i>Root XML Object</i>	No		Allows configuration of the namespace for the XSD into which this schema can be translated	first non-null
<code>definitions</code>	map<string, Definition Object>	No	Keys are the names of the definitions.	The schema type definitions for this schema	merged by key

Root XML object

Property	Type	Required	Format	Behavior	Combination Style
namespace	string	Yes		Must be a valid XSD namespace. Overrides the default namespace for the XSD with this value.	n/a

Definition object

Property	Type	Required	Format	Behavior	Combination Style
type	string	Yes	Must be object	Defines the type of JSON value to which this definition applies. Guidewire InsuranceSuite only supports definitions for JSON objects.	First non-null
title	string	No		Documentation only	First non-null
description	string	No		Documentation only	First non-null
properties	map<string, No Property Object>		The string keys in the map serve as property names. The values are the JSON schema definitions for those properties.	Defines the properties that this schema definition explicitly names. If a JSON object includes one of these specified properties, the names of the object must match the type and constraints set forth in the associated property definition.	Merged by key
additionalProperties	Property Object	No		This optional property indicates that the object can have arbitrary property values. Any property value that you do not explicitly name in the properties property must match the additionalProperties property definition.	Merged
required	string[]	No	Array of property names	JSON objects that conform to this definition must always specify the properties listed in the required property. If you do not specify the additionalProperties property, the names in the required property must all be valid named properties on this definition. Note that a property with a null value satisfies this constraint if the property specifies x-gw-nullable with a value of true. ClaimCenter reports a validation error only if the required property is undefined on the JSON object.	Merge names
x-gw-extensions	map<string, No any>		This property is an arbitrary map of property keys to values. The keys can be any string. The values can be any object, including JSON objects or arrays.	Values in extensions are available to IRestValidatorFactoryPlugin objects for use in building custom validations.	Merge extensions

Property object

Property	Type	Required	Format	Behavior	Combination Style
type	string	Either type or \$ref is required.	One of array, boolean, integer, number, or string	Defines the type of JSON value that this property is expected to have. If the property is a JSON object, set the \$ref property instead. The \$ref property indicates the expected object schema. For scalar types, the type, format, and x-gw-type properties define how data is serialized and deserialized. This combination also defines the mappings from JSON data to Java object types.	First non-null
title	string	No		Documentation only	First non-null
description	string	No		Documentation only	First non-null
format	string	No		The type, format, and x-gw-type properties define how data is serialized and deserialized. This combination also defines the mappings from JSON data to Java object types.	First non-null
x-gw-type	string	No		The type, format, and x-gw-type properties define how data is serialized and deserialized. This combination also defines the mappings from JSON data to Java object types.	First non-null
\$ref	string	Either type or \$ref is required.	Either #/definitions/<name> or <alias>#/definitions/<name>	Defines a reference to the schema definition to which this property generally conforms. The reference can be to another type defined within this schema. In the alternative, the reference can use an import alias to specify a definition from an imported file.	First non-null
items	Items Object	Required if type is array. Otherwise, the items property is not allowed.		Defines the items schema to which elements within this array must conform	Merged
default	any	No	Each element must be a JSON value that can be parsed based on the type, format, and x-gw-type of this property.	Defines a default value for this property in cases where the property is not defined on the input JSON object. The REST framework converts the default value as if it is a standard input value. The value is generally a valid JSON value for the type, format, and x-gw-type properties. For example, if the type property is string and the format property is gw-bigdecimal, the default value is generally a decimal string and not a JSON number. The default property is relevant only for scalar properties.	First non-null
maximum	number	No		Specifies the maximum value the property can have. The comparison is either inclusive or exclusive, depending upon the value of the exclusiveMaximum property. ClaimCenter parses the value of the maximum property as a fixed-point value with no loss of precision. ClaimCenter then converts the fixed-point value to the internal Java representation that the	First non-null

Property	Type	Required	Format	Behavior	Combination Style
				<p>runtime parameter uses for comparison purposes.</p> <p>Take the following scenarios for examples. On the one hand, suppose that the value of the <code>type</code> property is <code>integer</code> and the value of the <code>format</code> property is <code>int32</code>. In this case, the value of the <code>maximum</code> property must be a valid integer. ClaimCenter thus converts the value of the <code>maximum</code> property to an <code>Integer</code> at runtime.</p> <p>On the other hand, suppose that the respective values of the <code>type</code> and <code>format</code> properties are <code>string</code> and <code>gw-bigdecimal</code>. In this case, ClaimCenter converts the value of the <code>maximum</code> property to a <code>BigDecimal</code> for comparison at runtime. Note that ClaimCenter permits the <code>maximum</code> property to have a value only if the runtime type is numeric.</p>	
<code>exclusiveMaximum</code> boolean	boolean	No		Determines if the maximum value is treated as inclusive or exclusive. The default value of the <code>exclusiveMaximum</code> property is <code>false</code> . This means that the value of the <code>maximum</code> property is inclusive. ClaimCenter permits the <code>exclusiveMaximum</code> property to have a value only if you specify the <code>maximum</code> property.	First non-null
<code>minimum</code>	number	No		Specifies the minimum value the property can have. Otherwise, the <code>minimum</code> property behaves in the same way as the <code>maximum</code> property. Note that ClaimCenter permits the <code>minimum</code> property to have a value only if the runtime type is numeric.	First non-null
<code>exclusiveMinimum</code> boolean	boolean	No		Determines if the minimum value is treated as inclusive or exclusive. The default value of the <code>exclusiveMaximum</code> property is <code>false</code> . This means that the value of the <code>minimum</code> property is inclusive. ClaimCenter permits the <code>exclusiveMinimum</code> property to have a value only if you specify the <code>maximum</code> property.	First non-null
<code>maxLength</code>	integer	No		Determines the maximum length of the property value. The value of the maximum length is inclusive. For ClaimCenter to permit the <code>maxLength</code> property to have a value, the property must have a runtime type of <code>String</code> .	First non-null
<code>minLength</code>	integer	No		Determines the minimum length of the property value. The value of the minimum length is inclusive. For ClaimCenter to permit the <code>minLength</code> property to have a value, the property must have a runtime type of <code>String</code> .	First non-null
<code>pattern</code>	string	No		Specifies a regular expression that the property value must match. The regular expression is not implicitly anchored. If you want the expression to match the entire input, explicitly anchor the expression with <code>^</code> and <code>\$</code> . In this case, ClaimCenter evaluates the regular expression using the Java regular expression engine. The regular expression generally matches the Java	First non-null

Property	Type	Required	Format	Behavior	Combination Style
				syntax. This syntax has minor differences from the Javascript syntax and therefore represents a slight deviation from the JSON Schema specification. For ClaimCenter to permit the <code>pattern</code> property to have a value, the property must have a runtime type of <code>String</code> .	
<code>maxItems</code>	<code>integer</code>	No		Specifies the maximum number of elements allowed in an array. This value is inclusive. ClaimCenter permits <code>maxItems</code> for properties only of type <code>array</code> .	First non-null
<code>minItems</code>	<code>integer</code>	No		Specifies the minimum number of elements allowed in an array. This value is inclusive. ClaimCenter permits <code>minItems</code> for properties only of type <code>array</code> .	First non-null
<code>uniqueItems</code>	<code>boolean</code>	No		If set to <code>true</code> , this property indicates that the elements of the array must be unique. This uniqueness is as defined by the <code>equals()</code> and <code>hashCode()</code> methods of the deserialized values. The default value is <code>false</code> . ClaimCenter permits <code>uniqueItems</code> for properties only of type <code>array</code> .	First non-null
<code>multipleOf</code>	<code>number</code>	No		<p>Specifies that the property value must be a multiple of the specified value. ClaimCenter parses the value of the <code>multipleOf</code> property as a fixed-point value with no loss of precision. ClaimCenter then converts the fixed-point value to the internal Java representation that the runtime parameter uses for comparison purposes. Take the following scenarios for examples.</p> <p>On the one hand, suppose that the value of the <code>type</code> property is <code>integer</code> and the value of the <code>format</code> property is <code>int32</code>. In this case, the value of the <code>multipleOf</code> property must be a valid integer. ClaimCenter thus converts the value of the <code>multipleOf</code> property to an <code>Integer</code> at runtime.</p> <p>On the other hand, suppose that the respective values of the <code>type</code> and <code>format</code> properties are <code>string</code> and <code>gw-bigdecimal</code>. In this case, ClaimCenter converts the value of the <code>multipleOf</code> property to a <code>BigDecimal</code> for comparison at runtime. Note that ClaimCenter permits the <code>multipleOf</code> property to have a value only if the runtime type is numeric.</p>	First non-null
<code>readOnly</code>	<code>boolean</code>	No		Specifies that a given property value ought not be set for a JSON object that serves as an input to a REST API. The Swagger extensions to JSON Schema define the <code>readOnly</code> property. This property allows the same schema definition to be used for both input and output.	First non-null
<code>enum</code>	<code>any[]</code>	No	Each element must be a JSON value that can be parsed based	Specifies a list of values one of which the property value must match. ClaimCenter converts the enum values in the schema into Java objects at runtime. ClaimCenter then compares the enum values against the property	First non-null

Property	Type	Required	Format	Behavior	Combination Style
				on the type, format, and \$ref is required.	value using the equals and hashCode methods. x-gw-type of this property.
x-gw-xml	Property Object	XML	No	Allows configuration of how ClaimCenter maps the property definition into an XSD.	First non-null
x-gw-export-enumeration	boolean	No		If the value of this property is true, ClaimCenter writes typekey values out as an enum property while creating a schema for external clients. This property is only valid if enum is not set and x-gw-type is a typekey type. The default value is false.	First non-null
x-gw-nullable	boolean	No		Specifies that this property can have an explicit null value. This property defaults to false. A value of false means that the property must either be undefined on the object or must have an explicit non-null value.	First non-null
x-gw-extensions	map<string, Any>			This can be an arbitrary map of property keys to values. The keys can be any string. The values can be any object, including JSON objects or arrays.	Values in extensions are available to IRestValidatorFactoryPlugin objects for extensions use in building custom validations.
					Merge

Property XML object

Property	Type	Required	Format	Behavior	Combination Style
attribute	boolean	Yes		Specifies that a property is preferably mapped to an XML attribute rather than an XML element. The attribute property is not valid when x-gw-nullable has a value of true. The default value is false. In this case, schema properties are serialized as XML child elements.	n/a

Items object

Property	Type	Required	Format	Behavior	Combination Style
type	string	Either type or \$ref is required.	One of boolean, number, or string	The type, format, and x-gw-type properties define how array items are serialized and deserialized. This combination also defines the mappings from JSON data to Java object types.	First non-null
format	string	No		The type, format, and x-gw-type properties define how array items are serialized and deserialized. This combination also defines the mappings from JSON data to Java object types.	First non-null
x-gw-type	string	No		The type, format, and x-gw-type properties define how array items are serialized and	First non-null

Property	Type	Required	Format	Behavior	Combination Style
				deserialized. This combination also defines the mappings from JSON data to Java object types.	
\$ref	string	Either type or \$ref is required.	Either #/definitions/ <name> or <alias>#/definitions/ <name>	Defines a reference to the schema definition to which this property generally conforms. The reference can be to another type defined within this schema. In the alternative, the reference can use an import alias to specify a definition from an imported file.	First non-null
maximum	number	No		Specifies the maximum value property items are allowed to have. Otherwise, the behavior for the maximum property is identical to the behavior defined for the maximum property in the property object table.	First non-null
exclusiveMaximum	boolean	No		See the behavior for the exclusiveMaximum property in the property object table.	First non-null
minimum	number	No		Specifies the minimum value property items are allowed to have. Otherwise, the behavior for the minimum property is identical to the behavior defined for the minimum property in the property object table.	First non-null
exclusiveMinimum	boolean	No		See the behavior for the exclusiveMinimum property in the property object table.	First non-null
maxLength	integer	No		Determines the maximum length of property items. Otherwise, the behavior for the maxLength property is identical to the behavior defined for the maxLength property in the property object table.	First non-null
minLength	integer	No		Determines the minimum length of property items. Otherwise, the behavior for the minLength property is identical to the behavior defined for the minLength property in the property object table.	First non-null
pattern	string	No		Specifies a regular expression that property items must match. Otherwise, the behavior for the pattern property is identical to the behavior defined for the pattern property in the property object table.	First non-null
multipleOf	number	No		Specifies that the item values must be a multiple of the specified value. Otherwise, the behavior for the multipleOf property is identical to the behavior defined for the pattern property in the property object table.	First non-null
enum	any[]	No	Each element must be a JSON value that can be parsed based on the type, format, and x-gw-type of this item.	Specifies a list of values at least one of which item values must match. ClaimCenter turns the enum values in the schema into Java objects at runtime. ClaimCenter then compare the enum values against the item value using the equals and hashCode methods.	First non-null
x-gw-xml	Items XML Object	No		Allows configuration of how ClaimCenter maps the item definition into an XSD.	First non-null

Property	Type	Required	Format	Behavior	Combination Style
x-gw-export-enumeration	boolean	No		If the value of this property is true, ClaimCenter First non-null writes typekey values out as an enum property if creating a schema for external clients. This property is only valid if enum is not set and x-gw-type is a typekey type. The default value is false.	
x-gw-nullable	boolean	No		Specifies that this property can have items with an explicit null value. This property defaults to false.	First non-null
x-gw-extensions	map<string, No any>			This can be an arbitrary map of property keys use in building custom validations. to values. The keys can be any string. The values can be any object, including JSON objects or arrays.	Values in extensions are available to IRestValidatorFactoryPlugin objects for extensions Merge

Items XML object

Property	Type	Required	Format	Behavior	Combination Style
name	string	Yes		Must be a valid XML element or attribute name. Used for the name of XML elements that wrap each item in the list. By default, for an object array, the element name is the name of the referenced object definition. For scalar arrays, the default element name is the name of the containing property.	n/a

See also

- “JSON schema file combination” on page 844
- “JSON data types and formats” on page 847
- <https://tools.ietf.org/html/draft-zyp-json-schema-04>
- <https://tools.ietf.org/html/draft-fge-json-schema-validation-00>

JSON schema file combination

Combination overview

The combination process for JSON Schema files and the target use cases for it are similar to the process and use cases for Swagger schemas.

JSON schema combination specifics

Most of the differences between JSON Schema combination and Swagger schema combination involve two sets of specific details. These details include which elements are combined through which mechanism as well as how combinations are validated. Details about how you combine each element are listed in “JSON schema file specification” on page 836. The documentation does not cover explicitly how to validate combinations. However, validating the combination of JSON schemas follows the same philosophy as for validating the combination of Swagger schemas. Moreover, if A combines with B, the resulting document A' is a logical extension or superset of B. In addition, A' does not contain any destructive changes such as changes to the type of a property.

Structural comparisons

The primary characteristic that is unique to JSON Schema is how references are compared across versions. For example, suppose we have the schema `Base`:

```
"Foo" : {
  "properties" : {
    "bar" : { "$ref" : "#/definitions/Bar" }
  }
},
"Bar" : {
  "properties" : {
    "name" : { "type" : "string" }
  }
}
```

Suppose also that the schema `Ext` combines the schema `Base`. In addition, suppose that the schema `Ext` redefines the `bar` property to point to the `Baz` definition instead:

```
"Foo" : {
  "properties" : {
    "bar" : { "$ref" : "#/definitions/Baz" }
  }
},
"Baz" : {
  "properties" : {
    "name" : { "type" : "string" },
    "value" : { "type" : "string" }
  }
}
```

The validation algorithm for JSON schema combination declares this to be a legal change. The reason for allowing the change is because the validation algorithm compares the type references structurally rather than nominally. The distinction is that while deciding if two referenced types are compatible, the comparison algorithm ignores the names given to the schema definitions. Instead, the algorithm examines the structure that the definitions set forth.

In this case, the algorithm compares the `Baz` definition with the `Bar` definition. The algorithm determines whether `Baz` is a logical superset of `Bar`. `Baz` contains all the properties of `Bar`, and these properties have the same types. As such, the algorithm finds that substituting `Baz` in place of `Bar` is legal.

This structural comparison algorithm processes object and array object references recursively. Moreover, suppose that `Baz` and `Bar` both had properties that mapped to other objects or arrays of objects. In this case, the algorithm would compare each of these references structurally until the algorithm reaches the end of the tree.

The rough net effect of the structural comparison algorithm is to allow reference changes. This allowance is inclusive of reference changes that are inherent given import overrides. The net effect of this rule is that any data that validates at runtime against the old schema definition also validates against the new schema definition.

See also

- “JSON schema file specification” on page 836

JSON schema imports

Schema combination provides one mechanism for the reuse of schemas, by providing the ability to create a schema that logically extends and overrides one or more other schemas. The import mechanism gives you another way to reuse a schema, by creating a shared schema for common pieces and then importing that schema in order to reference its types.

Schema import and schema combine both provide units of reuse. A code analogy would be that the `combine` instruction is the equivalent of subclassing a given class while the `import` instruction is the equivalent of merely referencing that class in arguments or return types. Imports always exist in a separate logical namespace. If `A` imports `B`, definitions in `A` and `B` that have the same name are still two different types.

By contrast, schema combination exists in a single namespace. That is, if `A` combines `B`, definitions in `A` that have the same name as definitions in `B` are combined together to produce a single definition. From the point of view of downstream systems, everything ends up merged together anyway. Hence, the choice between the `import` instruction and the `combine` instruction really comes down to whether you want definitions with the same name to be merged together or not. When you are logically extending a schema, such as creating a custom schema that extends a base

configuration schema, you want the combination behavior. If you are referencing a shared schema authored by another team for another purpose, you probably do not want that behavior. In this case, keep the namespaces separate so that you do not have to worry about name conflicts producing unexpected results.

Import syntax

Imports in JSON schema files are done with the custom `x-gw-import` property. This property is an object where the keys are alias names and the values are fully-qualified JSON schema file names. References to types in the imported file are prefixed with the name of the alias before the `#` symbol. For example, the following `Contact` schema imports an `Address` schema:

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "x-gw-import": {
    "address": "gw.px.ete.address-1.0"
  },
  "definitions": {
    "Contact": {
      "type": "object",
      "properties": {
        "EmailAddress1": {
          "type": "string"
        },
        "HomePhone": {
          "type": "string"
        },
        "HomePhoneCountry": {
          "type": "string",
          "x-gw-type": "typekey.PhoneCountryCode"
        },
        "AllAddresses": {
          "type": "array",
          "items": {
            "$ref": "address#/definitions/Address"
          }
        },
        "PrimaryAddress": {
          "$ref": "address#/definitions/Address"
        }
      }
    }
  }
}
```

It is possible to override a schema import by combining schemas. For example, suppose a customer has a custom `Address` schema that defined extensions. In this case, the customer could define a `Contact` extension schema that overrides the `address` alias to point to the `Address` extension schema:

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "x-gw-combine": ["gw.px.ete.contact-1.0"],
  "x-gw-import": {
    "address": "customer.address-1.0"
  }
}
```

That override then causes all address references inherited from the base contact schema to resolve to definitions in the `customer.address-1.0` schema rather than the `gw.px.ete.address-1.0` schema.

Externalizing schemas

JSON schema files are often externalized either by running the `genExternalSchemas` command-line task or in the context of producing a Swagger schema that imports the JSON schema. When the files are externalized in this way, imports are automatically inlined to create a single schema file. Imports are also treated this way for XSD translation. The intention is to make any given schema self-contained for use by downstream tools. No optimal way exists to define JSON schema file imports in a stable way that all tools can understand.

While import mechanisms technically exist for XSDs, they are not always easy to operate with various XSD toolchains. As a result, imports exist as a tool for schema authors to break down their schemas into reusable chunks. However, schema consumers always see a unified view of the schema.

Imports can lead to naming conflicts. For example, the `Contact` schema could define its own type inline called `User`. At the same time, the `Address` schema could also have a type called `User`. The REST framework attempts to use

unqualified definition names in cases where there are no conflicts. However, the framework adds prefixes to ensure names are unique in cases where there are name conflicts. In this case, the output schema uses both `User` from the `Contact` schema itself and `address_User`, the definition from the `Address` schema.

JSON data types and formats

Supported formats

JSON natively defines a limited set of primitive types: `string`, `boolean`, `number` (double), `object`, and `array`. In addition, Swagger and JSON Schema define an `integer` type. The `integer` type is a number with a decimal component of `.0` or with no decimal component.

The limitations of the native JSON types naturally lead to questions about how to represent more complex data types. These complex data types include dates, numeric types that require a higher level of precision, binary data, and so forth.

To support such use cases, JSON Schema defines not only a `type` for properties but also a `format`. Swagger adopts those conventions for parameters as well. The `format` can convey additional information about the `type`. For example, the `format` can express that a string contains not just arbitrary string data but an ISO 8601-formatted date.

Swagger 2.0 defines standard formats that are part of the Swagger specification. Guidewire InsuranceSuite ClaimCenter adopts such formats where they are available. ClaimCenter also adds custom formats for data types that Swagger does not otherwise include. Such custom formats include fixed-point decimals—instances of the `BigDecimal` class in Java.

ClaimCenter also allows the `x-gw-type` property on Swagger parameters and JSON Schema properties. The `x-gw-type` property indicates a specific internal type to which ClaimCenter maps a property for serialization and deserialization. ClaimCenter currently uses the `x-gw-type` property for `typekey` types and to differentiate between `CurrencyAmount` and `MonetaryAmount` for money data types.

The combination of `type`, `format`, and `x-gw-type` properties determine three characteristics. These include the input Java types that ClaimCenter can serialize, the actual serialization format, and the Java type to which ClaimCenter deserializes the input data.

Types

The following table lists all the combinations of `type`, `format`, and `x-gw-type` that ClaimCenter understands as of release 10.0.0. The table also includes the standards in which you can find the various types:

Type	Format	x-gw-type	Standard
<code>boolean</code>	<code><none></code>	<code><none></code>	JSON Schema
<code>integer</code>	<code><none></code>	<code><none></code>	JSON Schema
<code>integer</code>	<code>int32</code>	<code><none></code>	Swagger
<code>integer</code>	<code>int64</code>	<code><none></code>	Swagger
<code>number</code>	<code><none></code>	<code><none></code>	JSON Schema
<code>number</code>	<code>float</code>	<code><none></code>	Swagger
<code>number</code>	<code>double</code>	<code><none></code>	Swagger
<code>string</code>	<code><none></code>	<code><none></code>	JSON Schema
<code>string</code>	<code>date-time</code>	<code><none></code>	Swagger
<code>string</code>	<code>date</code>	<code><none></code>	Swagger
<code>string</code>	<code>time</code>	<code><none></code>	Guidewire
<code>string</code>	<code>gw-bigdecimal</code>	<code><none></code>	Guidewire
<code>string</code>	<code>gw-biginteger</code>	<code><none></code>	Guidewire
<code>string</code>	<code>gw-money</code>	<code><none></code>	Guidewire

Type	Format	x-gw-type	Standard
string	gw-money	gw.pl.currency.MonetaryAmount	Guidewire
string	gw-money	gw.api.financials.CurrencyAmount	Guidewire
string	byte	<none>	Swagger
string	<none>	<any typekey>.* type	Guidewire

Java inputs and outputs for types

The following table lists the various types available for JSON, Swagger, and Guidewire schemas as well as the Java inputs and outputs that correspond to the types. In addition, the table contains examples of values and additional notes for the various types.

The **Java Inputs** column indicates which input types ClaimCenter can serialize to a given format. The **Java Output** column lists the output type to which ClaimCenter deserializes the data.

The input types are always a superset of the output type but are often more lenient. For example, the date type deserializes to a `LocalDate` but can serialize both `LocalDate` and `Date` types for the sake of convenience. In addition, the various numeric types can serialize lower magnitude numeric data types:

Type	Java Inputs	Java Output	Example	Notes
boolean	Boolean	Boolean	true, false	
integer	Byte, Short, or Integer	Integer	123	
integer	Byte, Short, or Integer	Integer	123	
integer	Byte, Short, Integer, Long or Long	Long	1234567890	Long objects in JSON are safe up to 15 digits without loss of precision (technically 2^{53}); for anything longer, use a string with <code>gw-biginteger</code> or <code>gw-bigdecimal</code> format.
number	Float or Double	Double	123.45	Numbers in JSON default to numbers; if you want to ensure you do not lose precision on decimal values (for example, on currency values), use a string type with <code>gw-bigdecimal</code> format.
number	Float	Float	123.45	
number	Float or Double	Double	123.45	
string	String	String	"test"	
string	Date	Date	"2017-07-24T16:37:16.123Z"	ISO 8601 date-time
string	Date or LocalDate	LocalDate	"2017-11-15"	ISO 8601 full-date
string	Date or LocalTime	LocalTime	"09:07:05.001"	ISO 8601 partial-time
string	BigDecimal	BigDecimal	"123456.00"	Formatted exactly with <code>BigDecimal.toPlainString()</code>
string	BigInteger	BigInteger	"12345678901234567890"	
string	MonetaryAmount	MonetaryAmount	"123.45 eur"	
string	CurrencyAmount	CurrencyAmount	"123.45 eur"	
string	Blob or byte[]	byte[]	"U29tZSB0ZXN0IHN0cmLuZw=="	Base64-encoded data
string	Specified typekey type	Specified typekey type	"USD"	Typekeys are serialized by code.

Note: Types that have primitive and object forms can have either of the forms as inputs. For example, the boolean type can have either a boolean primitive or a Boolean object as an input.

Default Java type mappings

For the purpose of serialization of `JsonObject` types without a schema, ClaimCenter defines default mappings from Java classes to JSON data types. The following table lists the default `type`, `format`, and `x-gw-type` for each supported Java type.

Java Type	JSON Type	JSON Format	JSON x-gw-type
Boolean	boolean	<none>	<none>
Integer	integer	int32	<none>
Long	integer	int64	<none>
Float	number	float	<none>
Double	number	double	<none>
String	string	<none>	<none>
Date	string	date-time	<none>
LocalDate	string	date	<none>
LocalTime	string	time	<none>
BigDecimal	string	gw-bigdecimal <none>	
BigInteger	string	gw-biginteger <none>	
CurrencyAmount	string	money	gw.api.financials.CurrencyAmount
MonetaryAmount	string	money	gw.pl.currency.MonetaryAmount
Blob	string	byte	<none>
byte[]	string	byte	<none>
any TypeKey type	string	<none>	typekey.<name>

See also

- <https://github.com/OAI/OpenAPI-Specification/blob/master/versions/2.0.md#data-types>
- <https://xml2rfc.tools.ietf.org/public/rfc/html/rfc3339.html#anchor14>

JSON parsing and validation

You can parse JSON data by invoking one of the `parse` methods on the `JsonObject` class or a generated `JsonWrapper` subclass. Parsing can be done in the context of a specific JSON schema definition. In this case, the data is validated and deserialized according to that schema. The data can also be parsed without a schema. In this case, no validation is performed. In addition, only a limited number of primitive types are used for deserialization. Deserialization and validation are done in a single step. A parser handles this step in order to maximize efficiency.

Schema-driven parsing

If a `JsonSchemaDefinition` is supplied to the parser, the REST framework uses the schema to drive deserialization and validation of the data.

Data types

The REST framework deserializes a given JSON property value based on an associated `JsonSchemaProperty` type:

- It deserializes object properties into `JsonObjects`.
- It deserializes scalar property values into the type determined by the properties `type`, `format`, and `x-gw-type`, as specified in “JSON data types and formats” on page 847.

- It deserializes array properties into lists of `JsonObjects` or scalar Java types, as appropriate.

Note: For the sake of client libraries written in JavaScript, values that map to a JSON integer type can have trailing decimal 0s. The REST framework still considers these values valid integers or longs. For example, suppose that a property is of type `integer` and that the property format is `int32`. In this case, the REST framework deserializes each of `100`, `100.0`, and `100.00` as the integer value `100`.

Constraint validation

JSON schema property definitions can define various constraints on a property value that can be used to validate the length of a property that deserializes as a `String`. For example, these constraints include the `maxLength` or `minLength` properties. Simple scalar constraints that are validated as property values are deserialized. Validation takes place against the deserialized values rather than the raw JSON values. Most of the validations are straightforward. Validation behavior is defined in “JSON schema file specification” on page 836. However, a few specific constraints have more complex behavior:

- If JSON data contains an explicit `null` value for a property or element within an array, the data is valid if and only if the properties or items involved declare `x-gw-nullable` to be `true`.
- If a schema definition defines required properties, these properties must be defined on the object; an explicit `null` value still counts as a property being defined for purposes of required property validation.
- Schema properties that specify `readOnly` as `true` are rejected if the deserialization options specify the `rejectReadOnlyProperties` option.
- Properties on the input that are not defined on the schema can either be rejected, logged, or ignored based on the value of the `unknownPropertyHandling` option in the deserialization options.

Custom validators

Custom validators for properties, items, and objects are invoked after the containing object has been fully parsed and deserialized.

Deserialization options

There are currently two different options that can be set while parsing and deserialization JSON data with the `JsonDeserializationOptions` class:

`unknownPropertyHandling`

This option determines what the framework does when it encounters properties on a JSON object that do not match a property in the schema. You can set this option to `ignore`, `log`, or `reject`, with `ignore` being the default value. The value is set automatically by the REST API framework, based upon the value of the `GW-UnknownPropertyHandling` request header or request property:

- If you set option `unknownPropertyHandling` to `ignore`, the framework deserializes the property values that do not match the schema and adds the property values to the resulting `JsonObject`.
- If you set option `unknownPropertyHandling` to `log`, the framework deserializes the property values still. In addition, the framework logs an INFO-level message.
- If you set option `unknownPropertyHandling` to `reject`, the framework reports a validation error for the property.

`rejectReadOnlyProperties`

This option determines whether the framework reports those properties whose schema specifies `readOnly` as `true` as validation errors. The default value is `false`. The REST API framework sets this value to `true` while parsing the body of a request.

Error reporting

Suppose a JSON object to be parsed is not well-formed JSON. That is, the JSON object cannot be parsed at all. In this case, an exception is thrown. Otherwise, any validation issues are reported back in the supplied

`JsonValidationResult` object. Note that many of the variants of the `JsonObject.parse` method implicitly ignore validation errors.

Schemaless parsing

If you do not supply a schema during JSON parsing, the framework uses the default rules to deserialize the JSON data. The resulting set of primitive data types is very limited and corresponds to the limited set of JSON primitive types.

Data types

JSON Type	Java Type
array	List
object	JsonObject
boolean	Boolean
integer	<ul style="list-style-type: none">• Integer if the integer is less than MAX_INT• Long if the integer is greater than or equal to MAX_INT and less than MAX_LONG• BigInteger if the integer is greater than MAX_LONG
number	Double
string	String

Error reporting

If the JSON to be parsed is not well-formed JSON, the framework throws an exception. Otherwise, the framework parsing completes successfully with no reported validation issues.

JSON serialization

It is possible to serialize out JSON data contained in a `JsonObject` to a JSON string by calling one of the `toJsonString` or `toPrettyJsonString` methods on the `JsonObject` class. As with JSON parsing, it is possible to perform JSON serialization with, or without, a schema.

Schema-driven serialization

JSON data serialized with a schema uses the schema to determine the output format for a property as well as to validate that the data conforms to the declared schema. Scalar value output formats are defined in “JSON data types and formats” on page 847. Handling of `null` values during serialization depends upon the serialization options that you are using.

Validation and error reporting

Schema-driven serialization can result in validation errors being reported in the following circumstances:

- The runtime value of a property in the `JsonObject` does not match the type of property defined within the schema. This circumstance can arise if a scalar value is present where a `JsonObject` or `List` is expected or if the scalar value is the wrong type of Java object.
- A property in the `JsonObject` does not match any property defined in the associated JSON schema definition, and the schema definition does not define the `additionalProperties` property.
- A property contains a `null` value, the property does not specify `x-gw-nullable` as `true`, and the serialization options specify the inclusion of `null` properties.

If any validation errors are found during serialization, the REST framework throws an exception.

Schemaless serialization

Serialization without a schema transforms `JsonObject` objects into JSON objects and lists of `JsonObject` objects into JSON arrays. For any other runtime value type, if the value type has a default JSON mapping as defined in “JSON data types

and formats” on page 847, the REST framework uses that JSON data type to serialize the value. Otherwise, if the framework cannot serialize the value, it reports a validation error, resulting in an exception.

Serialization Options

JSON serialization can accept a `JsonSerializationOptions` object, which has the following behavior. If the `toJsonString` or the `toPrettyJsonString` method is being invoked explicitly, the calling code can pass these options in directly. If the REST API framework invokes the method during serialization, ClaimCenter sets the serialization options based upon the values configured in the Swagger schema and runtime request headers. The following table summarizes the serialization options and how they behave:

Option	Default	Behavior
<code>includeNullProperties</code>	false	If true, null property values are preserved. If false, null properties are undefined in the output.
<code>includeNullItems</code>	false	If true, the REST framework preserves the null item values in arrays. If false, the REST framework omits null items from array values. In addition, the framework considers an array that contains only null items to be empty and handles the empty array depending upon the value of the <code>includeEmptyArrays</code> property.
<code>includeEmptyArrays</code>	true	If true, empty arrays for properties are preserved. If false, empty arrays are treated as if the property value was null, subject to the value of the <code>includeNullProperties</code> option.
<code>includeSchemaProperty</code>	false	Includes the <code>\$GW-Schema</code> property on the root object. This property includes the value of the fully-qualified schema definition name for the root object.
<code>includeCorrelationId</code>	false	Currently ignored for <code>JsonObject</code> serialization.
<code>includeTimestamp</code>	false	Currently ignored for <code>JsonObject</code> serialization.
<code>includeUser</code>	false	Currently ignored for <code>JsonObject</code> serialization.
<code>includeServer</code>	false	Currently ignored for <code>JsonObject</code> serialization.
<code>validateOutput</code>	false	If true, the REST framework validates the data against property-level constraints such as <code>minLength</code> , the set of required fields on the object, and any custom validators.

JsonObject class

The `JsonObject` class is the core representation of a JSON object within InsuranceSuite. The class can be used for both serialization and parsing / deserialization of JSON data. This use can be either with or without an explicit JSON Schema definition for the object. A `JsonObject` is essentially just a `Map<String, Object>` where every value in the Map is either a scalar value that our platform knows how to serialize, another `JsonObject`, or a `List` of scalars or `JsonObjects`. `JsonObjects` can be manipulated directly as `Maps` or can be optionally wrapped with statically-typed getters and setters. See the javadoc on the `JsonObject` class for details about the various helper methods for parsing and serializing data.

Schema versus schemaless

`JsonObjects` can be serialized or deserialized either with or without a JSON Schema. See “JSON parsing and validation” on page 849 and “JSON serialization” on page 851 for more details about how parsing and serialization work differently when a schema is supplied versus when one is not.

Late-binding of schemas

A `JsonObject` can work either with or without a schema. The purpose of this ability is to enable the late binding of REST API request and response schemas. Being “late-bound” allows for the extension of the core schemas by content or customer extensions.

For example, application code can be written against the schema `gw.cc.policy.policy-1.0`. The application code uses a `JsonObject` to represent input and output data for REST APIs that make use of definitions within the schema. A customer could then extend the schema to `customer.policy.policy-1.0` and extend the REST API to use the custom JSON schema. The end result is that the REST API framework knows what schema to use for input deserialization and output serialization—the customer schema.

However, the core application code does not need to be aware of the extension schema at all. In fact, the core application code does not even need to know the context in which it is being invoked. This strategic ignorance allows the same code to be shared across schema definitions for different minor versions of the same schema. This strategy also allows code sharing across for structurally similar but nominally distinct schema versions.

Keeping the `JsonObject` disconnected from the schema used to serialize or deserialize the object allows for code reuse and composition across layers. At the same time, the separation also allows the calling code ultimately to validate that the data conforms to the schema. Moreover, the separation allows the REST API framework to validate schema conformity.

JSON schema wrapper classes

Overview

The `JsonObject` class is a dynamic data structure—a Map of values. However, in cases where a JSON schema is defined, you might want to leverage the schema to provide statically typed getter and setter methods.

InsuranceSuite combines the benefits of a dynamic data structure with the benefits of static typing. To effect this combination, the software can generate statically typed classes that wrap a `JsonObject` class in a layer of statically-typed getters and setters. Once a wrapper has been layered on a `JsonObject`, you can access the object as if it were a DTO object into which data is serialized.

Such statically typed wrapper classes always extend the base `JsonWrapper` class. They also have typed getters and setters for all properties declared in the schema. In addition, the wrapper classes have helper methods for parsing in a schema-aware way. The REST API framework is integrated such that you can use the `JsonWrapper` subclasses anywhere that you can use a `JsonObject`. The framework then automatically wraps or unwraps the `JsonObject` object as needed.

Package and class naming

ClaimCenter generates the schema wrapper classes in Java under the `jsonschema` namespace. ClaimCenter generates the classes in a package named `jsonschema.<schema package>.<name>.v<schema version>`.

For example, the schema `gw.pc.activities.activities-1.0` would have wrapper classes generated under `jsonschema.gw.pc.activities.activities.v1_0`. Each definition in the schema then has a generated class with a name that matches the definition name. Package, class, and getter/setter names that would otherwise be illegal Java identifiers are adjusted as necessary to make them legal.

Wrapping and unwrapping

A new wrapper class can be instantiated with an empty `JsonObject` by calling the no argument constructor on the wrapper class. A wrapper class can be applied to an existing `JsonObject` by calling the static `wrap` method on the class.

The static `wrap` method always preserves `null` properties. Moreover, calling `wrap(null)` returns `null` back. For example:

```
var json : JsonObject
var activityDetail = ActivityDetail.wrap(json)
```

Turning the wrapper back into a `JsonObject` is as simple as calling the `unwrap` method on a `JsonWrapper`:

```
function toJson(activity : Activity) : JsonObject {
    var activityDetail = new ActivityDetail() // Creates a new, empty JsonObject and wraps it with an ActivityDetail()
    activityDetail.subject = activity.Subject
    return activityDetail.unwrap()
}
```

Parse helpers

`JsonWrapper` classes have a pair of static `parse` methods that can be used to parse JSON strings. The methods use the same schema definition that was used to generate the classes. These can be helpful in situations like tests. For

example, you can invoke a REST API first. Then, you can parse the response directly into a wrapper. Lastly, you can easily use this wrapper in test assertions.

Parsing this way violates the *late-binding* principle mentioned in “`JsonObject` class” on page 852. However, this manner of parsing can be helpful when debugging test code. This parsing method is also ideal for developing customer-specific code in that the developer does not have to consider extending the underlying schema.

Additional properties

If a JSON schema definition specifies `additionalProperties`, the generated wrapper class has `getAdditionalProperty` and `setAdditionalProperty` methods. These methods essentially provide a statically-typed wrapper on top of a basic `Map` `get` or `put`.

Lists and subobjects

Subobjects retrieved with the statically-typed helpers are themselves wrapped on retrieval. These wrappers are not stable. Rather, they are created for each individual call. Likewise, when statically-typed helpers are invoked to set a subobject, they implicitly unwrap the wrapper and set the underlying `JsonObject` as the value of the property. In general, the lack of stability of the `JsonWrapper` references is not a problem as the underlying `JsonObject` contains the data and is always the same.

However, lists of subobjects present a somewhat more awkward case. While retrieving a `List` of subobjects, the underlying `List` is wrapped in a special list that dynamically wraps each object as it is retrieved. The list also unwraps `JsonWrapper` objects as they are added to the `List` object. When setting a `List` of subobjects, a new `List` has to be created and set on the underlying `JsonObject`. The values of the original list are unwrapped and replaced. Thus, reading the values of a list is straightforward and looks analogous to the equivalent operation with a normal DTO-style class. However, while setting a `List` value, modifications to the original `List` object are not reflected in the underlying `JsonObject`. For example, the following code works as expected:

```
var contact : ContactDetail
var addresses = new List<AddressDetail>()
addresses.add(new AddressDetail() { :addressLine1 = "100 Main St." })
addresses.add(new AddressDetail() { :addressLine1 = "200 Main St." })
contact.addresses = addresses
```

But this code fails because calling `contact.addresses = addresses` implicitly copies the `List` object. The second call to `add` is not reflected in the `ContactDetail` `JsonObject`:

```
var contact : ContactDetail
var addresses = new List<AddressDetail>()
addresses.add(new AddressDetail() { :addressLine1 = "100 Main St." })
contact.addresses = addresses
addresses.add(new AddressDetail() { :addressLine1 = "200 Main St." })
```

There are three ways to avoid this problem:

- Make sure to add all elements to a newly constructed list prior to calling the setter on the wrapper object.
- Re-retrieve the list from the wrapper object for subsequent additions rather than using the original list.
- Use the `addToX` helper method that is generated for each list.

The `addToX` helper methods have an additional helpful quality in that they create the underlying `List` object if the property is `null`. This code can produce a `NullPointerException`:

```
var contact : ContactDetail
contact.addresses.add(new AddressDetail() { :addressLine1 = "100 Main St." })
```

While this code avoids that problem:

```
var contact : ContactDetail
contact.addToAddresses(new AddressDetail() { :addressLine1 = "100 Main St." })
```

Lists of scalar values do not present the same difficulty and do not require wrapping. However, they still benefit from the `addToX` helper method to auto-create the backing `List`.

Generating the classes

Unlike our existing code generators, the JSON schema wrapper classes must be generated explicitly. Then, the classes must be checked in to source code rather than being automatically generated as part of the build. This check-in requirement is partly a pragmatic concession. The concession is due to the level of effort required in order to build a bullet-proof code generator that is properly incremental and fast enough to run as part of the build. The check-in requirement is also partly a deliberate design decision. The decision gives Guidewire more freedom to make the wrapper classes optional or to change the code generator over time without impacting existing uses.

In order to generate the classes, add the fully-qualified name of the schema to generate classes to `config/integration/schemas/codegen-schemas.txt`.

The code generator can be invoked either from an IntelliJ run command or from the command line:

- In IntelliJ, you can add a run command for `com.guidewire.tools.json.JsonSchemaWrapperCodegenTool`.
- From the command-line, you can run the `jsonSchemaCodegen` task.

One unfortunate pitfall is that you can sometimes get into a state in which running the code generator can leave code in a non-compiling state. In this case, you need to get back to a compiling state before you can run the code generator to fix the problem. Using source control to revert the generated classes to their previous state is generally the best option to return to a compiling state.

XML output and XSD translation

XML output

The `JsonObject` and `TransformResult` classes both support serialization to XML rather than serialization to JSON. In both cases, exactly the same serialization and validation rules apply to serializing out to XML as to JSON. Only the output format differs.

- To serialize a `JsonObject` to XML, invoke one of the `toXmlElement` methods and then serialize the `XmlElement` as desired.
- To serialize a `TransformResult` to XML, invoke one of the `toXmlElement` methods. Then, serialize the `XmlElement` as desired. In the alternative, call one of the `toXmlString` convenience methods.

In addition, suppose that a REST API handler class returns a `TransformResult` or `JsonObject` object. In this case, the REST API framework automatically serializes the data to XML if the negotiated content type is `application/xml` or `text/xml`.

XSD translation

The XML that is output from serializing a `JsonObject` object with a schema, or `TransformResult` object (which always has a schema), conforms to the XSD that is produced as a transformation on the JSON Schema. In general, the translation to an XSD follows the following rules:

- The generated XSD has a single namespace of `http://guidewire.com/xsd/<schema-fqn>` by default. You can override this default by specifying the `x-gw-xml` property on the root of the JSON schema and setting the `namespace` attribute.
- All JSON schema definitions are translated into XSD `complexType`s as well as top-level elements that reference those types. That is, any such element is a legal document root.
- By default, definition properties are mapped to sub-elements rather than attributes. That is unless the schema property specifies the `x-gw-xml` element and sets the `attribute` property on the element to `true`.
- Properties under an element are unordered in the XML. The properties receive an `xs:all` in the XSD rather than an `xs:sequence`.
- The element or attribute name for a property name has the same name in an XSD as a JSON schema property. The REST framework adjusts the name if it is not a legal XML element or attribute name.
- The frame work always wraps arrays in a container element with individual elements of the array wrapped in elements. If the item elements are objects, they default to having a name based on the referenced object definition.

Alternatively, if the item elements are scalars, they have the same name as the containing property. It is possible to specify the `x-gw-xml` element on the items and give a `name` property to explicitly name the child elements.

- The framework maps JSON data types to equivalent XSD data types when available.
- The framework reproduces validation constraints such as `minLength` or `maxLength` on the property in the XSD whenever possible.
- The framework maps a schema definition that contains `additionalProperties` into an XSD if no explicitly-named properties are defined on the definition. However, the schema definition is represented as an `xs:any` if `additionalProperties` is mixed with explicitly-named properties.

Generating the XSDs

XSDs for JSON schema documents are produced using the standard tool that produces externalized schemas. That is the `gwb genExternalSchemas` task.

Example

Consider the following JSON schema document:

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "definitions": {
    "Contact": {
      "type": "object",
      "properties": {
        "AllAddresses": {
          "type": "array",
          "items": {
            "type": "object",
            "$ref": "#/definitions/Address"
          }
        },
        "EmailAddress1": {
          "type": "string"
        },
        "FirstName": {
          "type": "string"
        },
        "HomePhone": {
          "type": "string"
        },
        "HomePhoneCountry": {
          "type": "string",
          "x-gw-type": "typekey.PhoneCountryCode"
        },
        "IntegerExt": {
          "type": "integer",
          "format": "int32"
        },
        "LastName": {
          "type": "string"
        },
        "OfficialIDs": {
          "type": "array",
          "items": {
            "$ref": "#/definitions/OfficialID"
          }
        },
        "PrimaryAddress": {
          "$ref": "#/definitions/Address"
        },
        "TagTypes": {
          "type": "array",
          "items": {
            "type": "string",
            "x-gw-type": "typekey.ContactTagType"
          }
        }
      }
    },
    "OfficialID": {
      "type": "object",
      "properties": {
        "Jurisdiction": {
          "type": "string",
          "x-gw-type": "typekey.Jurisdiction"
        },
        "Type": {
          "type": "string",
          "x-gw-type": "typekey.OfficialIDType"
        }
      }
    }
  }
}
```

```

        "Value": {
            "type": "string"
        }
    },
    "TagMap": {
        "type": "object",
        "additionalProperties": {
            "type": "string",
            "x-gw-type": "typekey.ContactTagType",
            "x-gw-export-enumeration": true
        }
    },
    "Address": {
        "type": "object",
        "properties": {
            "AddressLine1": {
                "type": "string"
            },
            "AddressLine2": {
                "type": "string"
            },
            "City": {
                "type": "string"
            }
        }
    },
    "properties": {
        "Contact": {
            "type": "object",
            "$ref": "#/definitions/Contact"
        },
        "OfficialID": {
            "type": "object",
            "$ref": "#/definitions/OfficialID"
        },
        "TagMap": {
            "type": "object",
            "$ref": "#/definitions/TagMap"
        },
        "Address": {
            "type": "object",
            "$ref": "#/definitions/Address"
        }
    }
}

```

That produces the following XSD:

```

<?xmlversion="1.0"?>
<xss:schema
    targetNamespace="http://guidewire.com/xsd/gw.px.ete.contact-1.0"
    version="1.0"
    elementFormDefault="qualified"
    xmlns="http://guidewire.com/xsd/gw.px.ete.contact-1.0"
    xmlns:gw="http://guidewire.com/xsd"
    xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xss:complexType
        name="Contact">
        <xss:all>
            <xss:element
                name="AllAddresses"
                minOccurs="0">
                <xss:complexType>
                    <xss:sequence>
                        <xss:element
                            name="Address"
                            type="Address"
                            minOccurs="0"
                            maxOccurs="unbounded"/>
                    </xss:sequence>
                </xss:complexType>
            </xss:element>
            <xss:element
                name="EmailAddress1"
                type="xs:string"
                minOccurs="0"/>
            <xss:element
                name="FirstName"
                type="xs:string"
                minOccurs="0"/>
            <xss:element
                name="HomePhone"
                type="xs:string"
                minOccurs="0"/>
            <xss:element
                name="HomePhoneCountry"
                type="xs:string"

```

```
    minOccurs="0"
    gw:type="typekey.PhoneCountryCode"/>
<xs:element
    name="IntegerExt"
    type="xs:int"
    minOccurs="0"/>
<xs:element
    name="LastName"
    type="xs:string"
    minOccurs="0"/>
<xs:element
    name="OfficialIDs"
    minOccurs="0">
    <xs:complexType>
        <xs:sequence>
            <xs:element
                name="OfficialID"
                type="OfficialID"
                minOccurs="0"
                maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element
    name="PrimaryAddress"
    type="Address"
    minOccurs="0"/>
<xs:element
    name="TagTypes"
    minOccurs="0">
    <xs:complexType>
        <xs:sequence>
            <xs:element
                name="TagTypes"
                type="xs:string"
                minOccurs="0"
                maxOccurs="unbounded"
                gw:type="typekey.ContactTagType"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
</xs:all>
</xs:complexType>
<xs:complexType
    name="OfficialID">
    <xs:all>
        <xs:element
            name="Jurisdiction"
            type="xs:string"
            minOccurs="0"
            gw:type="typekey.Jurisdiction"/>
        <xs:element
            name="Type"
            type="xs:string"
            minOccurs="0"
            gw:type="typekey.OfficialIDType"/>
        <xs:element
            name="Value"
            type="xs:string"
            minOccurs="0"/>
    </xs:all>
</xs:complexType>
<xs:complexType
    name="TagMap">
    <xs:sequence>
        <xs:any
            minOccurs="0"
            maxOccurs="unbounded"
            processContents="skip"/>
    </xs:sequence>
</xs:complexType>
<xs:complexType
    name="Address">
    <xs:all>
        <xs:element
            name="AddressLine1"
            type="xs:string"
            minOccurs="0"/>
        <xs:element
            name="AddressLine2"
            type="xs:string"
            minOccurs="0"/>
        <xs:element
            name="City"
            type="xs:string"
            minOccurs="0"/>
    </xs:all>
</xs:complexType>
<xs:element
    name="Contact"
```

```
    type="Contact"/>
<xss:element
  name="OfficialID"
  type="OfficialID"/>
<xss:element
  name="TagMap"
  type="TagMap"/>
/<xss:schema>
```

The following is the corresponding JSON output:

```
{
  "AllAddresses": [
    {
      "AddressLine1": "100 Main St."
    }
  ],
  "FirstName": "Alice",
  "LastName": "Applegate"
}
```

The JSON output would like the following when serialized to XML:

```
<Contact xmlns="http://guidewire.com/xsd/gw.px.ete.contact-1.0">
  <AllAddresses>
    <Address>
      <AddressLine1>100 Main St.</AddressLine1>
    </Address>
  </AllAddresses>
  <FirstName>Alice</FirstName>
  <LastName>Applegate</LastName>
</Contact>
```

Releasing schemas

A publisher of an API for external consumption might want to ensure against the accidental deployment of changes into production environments. This aim would especially apply to changes that modify a JSON or Swagger schema unintentionally. In order to detect API changes, Guidewire InsuranceSuite allows you to mark a given schema as *released*. This marking means that you do not expect further changes to a schema version.

Marking a schema as *released* causes ClaimCenter to generate a checksum for the schema under `config/integration/schemas/versions.json`. Suppose a schema file is loaded at runtime and `versions.json` lists the schema file. In this case, the system generates a checksum for the schema and compares the checksum against a checksum for the stored version. If the schema version does not match and the server is in production mode, the server does not start. Otherwise, ClaimCenter logs an error message.

Excluded properties

The hashing algorithm for schema files removes **documentation-only** properties. For a JSON schema, this removal applies to the `description` property. For a Swagger schema, the hashing also excludes `info`, `description`, `summary`, `example`, and `externalDocs` properties. In addition, the order of properties within a given object does not affect the resulting hash.

Releasing a schema

It is possible to use the `updateReleasedSchemaVersions` command-line task to add a schema to `versions.json` or to update the checksum of a released schema. For example:

```
gwb updateReleasedSchemaVersions --module configuration --schemaType schema --versions
  gw.cc.activities.activities-1.0
```

This code adds the `activities-1.0` JSON schema from the `cc` module.

Externalized JSON schemas

JSON schemas authored within InsuranceSuite make use of a number of Guidewire-specific extensions. These extensions are especially prominent around composition and extension of schemas.

External systems naturally are not able to consume JSON schema documents directly as they are. The systems do not understand the particular properties in the schemas. To consume a JSON schema from another system, you must produce an externalized version of the schema that standard third-party tools can understand.

Externalizing a JSON schema:

- Produces a combined version of the schema. In the combined version, all `x-gw-combine` statements have been processed.
- Inlines any imported schema definitions and rewrites `$ref` properties appropriately to point to the inlined definitions.
- Removes custom `x-gw-` properties that are not relevant for downstream systems.
- Completes a few other small transformations. Such transformations include adding `enum` to properties that specify `x-gw-export-enumeration`.
- Adds top-level properties to the schema to reference every definition. This addition is because many third-party tools expect that a schema to represent a single object definition. The third-party tools cannot produce generated classes for any embedded definitions that appear unreferenced.

An analogous process happens when producing XSDs for a given JSON schema and when producing external Swagger schemas. This process incorporates the JSON schema.

Command-line tools

To generate the externalized schemas, run the `genExternalSchemas` build task. By default, the tool generates externalized versions of all defined schemas. However, you can use command-line arguments to generate specific schemas.