



Guidewire PolicyCenter™

Integration Guide

Release 10.0.3

© 2019 Guidewire Software, Inc.

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

Guidewire Proprietary & Confidential — DO NOT DISTRIBUTE

Product Name: Guidewire PolicyCenter

Product Release: 10.0.3

Document Name: Integration Guide

Document Revision: 12-December-2019

Contents

Guidewire Documentation	22
Guidewire InsurancePlatform™	22
About PolicyCenter documentation	22
Support	24

Part 1

Planning integration projects

1 Overview of integration methods	27
PolicyCenter integration elements	30
Preparing for integration development	30
Overview of integration documentation for programmers	31
Regenerating integration libraries and WSDL	32
Regenerating the Java API libraries	32
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	34
Guidewire internal classes and methods	35

Part 2

Web services

2 Overview of web services	39
Publishing or consuming web services from Gosu	39
What happens during a web service call	40
Reference of all base configuration web services	40
3 Publishing web services	43
Web service publishing quick reference	43
Web service publishing annotation reference	45
Data transfer objects	46
Publishing and configuring a web service	47
Declaring the namespace for a web service	47
Specifying the minimum run level for a web service	48
Specifying required permissions for a web service	48
Overriding a web service method name or visibility	49
Web service invocation context	49
Web service class life cycle and variables scoped locally to a request	51
Generating and publishing WSDL	51
Example WSDL	52
Calling a PolicyCenter web service from Java	53
Generate Java classes that make SOAP client calls to a SOAP API	54
Using the generated Java classes	55
Adding HTTP basic authentication in Java	55
Adding SOAP header authentication in Java	56
Testing web services with local WSDL	57

Using your class to test local web services	57
Writing unit tests for your web service	58
Web services authentication plugin	58
Writing an implementation of the web services authentication plugin	59
Login authentication confirmation	59
Request or response XML structural transformations	60
Transforming a generated schema	60
Converting a Gosu object to and from XML	61
Using predefined XSD and WSDL	63
Adding an invocation handler for preexisting WSDL	63
Example of an invocation handler for preexisting WSDL	63
Invocation handler responsibilities	65
Referencing additional schemas in your published WSDL	66
Validating requests using additional schemas as parse options	67
Creating an XML Schema JAR file	67
Setting response serialization options, including encodings	70
Adding advanced security layers to a web service	70
Transformations on data streams	70
Using WSS4J for encryption, signatures, and other security headers	72
Locale and language support	75
Checking for duplicate external transaction IDs	75
Exposing typelists and enums as enumeration values and string values	76
Optional stateful session affinity using cookies	76
4 Calling web services from Gosu	79
Loading WSDL locally by using web service collections	80
Loading WSDL directly into the file system for testing purposes	80
Types of client connections	81
How Gosu processes WSDL	81
Web service API objects are not thread safe	82
Working with web service data by using Gosu XML APIs	83
What Gosu creates from your WSDL	83
Special behavior for multiple ports	83
Request XML complexity affects appearance of method arguments	84
Adding configuration options to a web service	84
Directly modifying the WSDL configuration object for a service	85
Authentication schemes for consuming WS-I web services from Gosu	86
Guidewire suite configuration file	88
Setting Guidewire transaction IDs on web service requests	89
Setting a timeout on a web service	90
Custom SOAP headers	90
Server override URL	90
Setting XML serialization options	90
Setting locale or language in a Guidewire application	91
Implementing advanced web service security with WSS4J	91
One-way methods	92
Asynchronous method calls to web services	93
MTOM attachments with Gosu as web service client	93
5 General web services	95
Importing administrative data	95
Prepare exported administrative data for import	95
Importing prepared administrative data	95
CSV import and conversion	96
Advanced import or export	96
Maintenance tools web service	96

Using web service methods with batch processes	96
Using web service methods with work queues	97
Using web service methods with startable plugins	99
Mapping typecodes and external system codes	99
The TypeListToolsAPI web service	100
The TypecodeMapper class	101
The Profiler API web service	102
System tools web service	104
Getting and setting the run level	105
Getting server and schema versions	105
Clustering tools	105
Table import web service	107
Template web service	109
Workflow web service	110
Zone data import web service	111
6 Account and policy web services	113
Date- and time-related DTOs	113
Account web services	114
Add documents to an account	114
Add notes to an account	114
Find accounts	114
Assign activities	114
Add contacts to an account	114
Add location to an account	115
Retrieve account numbers	115
Check for active policies	115
Insert accounts	115
Merge accounts	115
Transfer policies	115
Job web services	116
Common parameters in the job APIs	116
Helper methods of the job APIs	116
Policy cancellation and reinstatement web services	116
Begin a cancellation	117
Rescind a cancellation	117
Find a cancellation	118
Reinstate a policy	118
Submission web services	119
Passing in external policy period data for submissions	119
Start a submission	119
Start a submission and generate a quote	119
Producer web services	120
Product Model web services	120
Synchronize Product Model in database with file system XML	120
Synchronize system tables	121
Query the database for Product Model information	121
Policy web services	122
Add a referral reason	122
Close a referral reason	122
Add activity from pattern	123
Policy period web services	124
Add notes to policies and policy periods	124
Add documents to policy periods	124
Policy earned premium web services	124

Import policy web services	125
Policy change web services	125
Starting an automatic policy change job	125
Starting a manual policy change job	126
Specifying what policy data to change	126
Policy renewal web services	126
Start renewals on existing policies	126
Import a policy for renewal	126
Policy renewal methods for billing systems	127
Archiving web services	128
Check if a policy term is archived	128
Restore a policy term	128
Suspend and resume archiving for a policy	128
Quote purging web services	129
Do not purge flag on a policy period	129

Part 3

Plugins

7 Overview of plugins	133
Implementing plugin interfaces	134
Choose a plugin implementation type	134
Writing a plugin implementation class	134
Registering a plugin implementation class	136
Deploying Java files, including Java code called from Gosu plugins	137
Error handling in plugins	138
Enable or disable a plugin	138
Example Gosu plugin	138
Special notes for Java plugins	139
Getting plugin parameters from the plugins registry editor	139
Getting the local file path of the root and temp directories	139
Plugin registry APIs	139
Plugin thread safety	140
Using Java concurrent data types, even from Gosu	141
Avoiding singletons because of thread-safety issues	142
Design plugin implementations to support server clusters	143
Reading system properties in plugins	143
Do not call local web services from plugins	144
Creating unique numbers in a sequence	144
Restarting and testing tips for plugin developers	144
Summary of PolicyCenter plugins	144
8 Account and policy plugins	153
Account plugin	153
Audit schedule selector plugin	154
Conversion on renewal plugin	155
Effective time plugin	155
ETL Product Model Loader plugin	157
Job number generator plugin	158
Job process creation plugin	158
Location plugin	159
Loss history plugin	160
Motor vehicle record (MVR) plugin	160
How PolicyCenter uses motor vehicle records	161
Notification plugin	162

Payment gateway configuration plugin	163
Payment gateway plugin	164
Policy evaluation plugin	165
Policy hold job evaluation plugin	166
Policy number generator plugin	167
Policy payment plugin	167
Policy period plugin	168
Policy plugin	171
Policy term plugin	174
Proration plugin	176
Add a plugin parameter to the proration plugin	176
Provide your own prorater subclass	176
Quote purging plugin	178
Reference date plugin	181
Renewal plugin	181
Underwriting company plugin	182
9 Authentication integration	185
Overview of user authentication interfaces	185
User authentication source creator plugin	186
Handling errors in the authentication source plugin	187
Create a custom exception type	187
Authentication data in HTTP attributes	187
Authentication data in parameters in the URL	188
Authentication data in HTTP headers for HTTP basic authentication	188
User authentication service plugin	189
Authentication service sample code	190
Handling errors in authentication service plugins	190
SOAP API user permissions and special-casing users	191
Example authentication service authentication	191
Deploying user authentication plugins	192
Database authentication plugins	192
Configuration for database authentication plugins	194
WS-I web services authentication	194
10 Document management	195
Document storage overview	195
Storing document content and metadata	196
Document storage plugin architecture	196
Understanding document IDs	198
Document IDs in emails and notes	198
Implementing a document content source for external DMS	198
Check for document existence	199
Add documents and metadata	200
Retrieve documents	201
Update documents and metadata	203
Remove documents	203
User interface elements when document management system unavailable	203
Render a document	204
Implementing an IDocumentMetadataSource plugin	204
Basic methods for the IDocumentMetadataSource plugin	205
Linking methods for the IDocumentMetadataSource plugin	207
Document storage plugins in the base configuration	208
Documents storage directory and file name patterns	208
Remember to store public IDs in the external system	209
Asynchronous document storage	209

Configure asynchronous document storage (sync-first or async-only)	213
Disable asynchronous document content storage completely (sync-only)	214
Errors and document validation in asynchronous document storage	214
Final documents	215
Final documents with IDocumentMetadataSource enabled	215
Final documents with IDocumentMetadataSource disabled	215
Document interactions also depend on user permissions	216
APIs to attach documents to business objects	216
Gosu APIs to attach documents to business objects	216
Web service APIs to attach documents to business objects	216
11 Document production	217
Document production types	218
Understanding synchronous and asynchronous document production	218
User interface flow for document production	218
Document production plugins	221
Implementing synchronous document production	222
Implementing asynchronous document production	223
Configuring document production implementation mapping	223
Add a custom MIME type for document production	224
Licensing for server-side document production	224
Licensing for Microsoft Excel and Word document production	224
Licensing for PDF document production	225
Write a document template descriptor and install a template	225
Example simple template descriptor XML file with no context objects	226
Example template descriptor XML file with context objects	227
Form fields and field groups	229
Field groups	230
Display values	230
Context objects	232
Gosu APIs on template descriptor instances	234
Template descriptor fields related to form fields and values to merge	235
XML format for document template descriptors	235
Add custom attributes document template descriptor XML format	236
Create your actual template file	237
Adding document templates and template descriptors to a configuration	237
Creating new documents from Gosu rules	238
Example document creation after sending an email	239
Important notes about cached document UIDs	239
12 Geographic data integration	241
Geocoding plugin integration	241
How PolicyCenter uses geocode data	241
What the geocoding plugin does	241
Synchronous and asynchronous calls to the geocoding plugin	242
Using a proxy server with the geocoding plugin	242
Batch geocoding only some addresses	242
Built-in Bing maps geocoding plugin	243
Geocoding and routing responses	243
Geocoding request generators	243
Deploy a geocoding plugin	244
Writing a geocoding plugin	245
Using the abstract geocode Java class	245
High-level steps to writing a geocoding plugin implementation	245
Geocoding an address	246
Getting driving directions	247

Getting a map for an address	249
Getting an address from coordinates (reverse geocoding)	249
Geocoding status codes	250
List of geocoding status codes	250
13 Encryption integration	251
Using encrypted properties	251
Setting encrypted properties	251
Querying encrypted properties	252
Sending encrypted properties to other systems	253
Setting up encryption	254
Writing your encryption plugin	254
Example encryption plugin	255
Changing your encryption algorithm	257
Change your encryption algorithm	257
Installing your encryption plugin	258
14 Management integration	259
Management plugin examples	260
Calling the example JMXManagementPlugin	261
15 Other plugin interfaces	263
SharedBundlePlugin marker interface	263
Credentials plugin	263
Preupdate handler plugin	264
Validation plugin	265
Work item priority plugin	266
Exception and escalation plugins	266
Sending emails	266
IEmailTemplateSource plugin	269
Defining base URLs for fully qualified domain names	269
Vehicle identification number plugin	271
Automatic address completion and fill-in plugin	271
Phone number normalizer plugin	271
Official IDs mapped to tax IDs plugin	272
Testing clock plugin (for non-production servers only)	273
Using the testing clock	273
16 Startable plugins	275
Starting a startable plugin	275
Initializing a startable plugin	276
Registering startable plugins	276
Manually starting and stopping a startable plugin	276
Writing a singleton startable plugin	276
User contexts for startable plugins	278
Simple startable plugin example	278
Writing a distributed startable plugin	279
Defining startable plugins in Java	281
Persistence and startable plugins	281
17 Multi-threaded inbound integration	283
Inbound integration core plugin interfaces	283
Inbound integration handlers for file and JMS integrations	284
Configuring inbound integration	285
Thread pool configuration XML elements	286
Varying the inbound integration settings	287

Inbound integration configuration XML elements	287
Using the inbound integration polling and throttle intervals	291
Inbound file integration	292
Create an inbound file integration	292
Example of an inbound file integration	294
Inbound JMS integration	295
Create an inbound JMS integration	295
Example of an inbound JMS integration	297
Custom inbound integrations	298
Writing a custom inbound integration plugin	299
Writing a work agent implementation	300
Install a custom inbound integration	304

Part 4

Messaging

18 Messaging and events	309
Overview of messaging	309
Event	309
Message	309
Message history	310
Messaging destinations	310
Root object	310
Primary entity and primary object	310
Acknowledgment	311
Safe ordering	311
Transport neutrality	311
Overview of messaging flow	311
Messaging flow details	312
Overview of message destinations	315
Use the Messaging editor to create new messaging destinations	317
Sharing plugin classes across multiple destinations	319
Handling acknowledgments	319
Destinations in the base configuration	320
Message processing	320
Message processing cycle	320
Messaging database transactions during sending	322
Messaging plugin interaction and flow diagram	328
Messaging events in PolicyCenter	329
List of messaging events	330
Triggering a remove-related event	337
Triggering custom events	337
Custom events from SOAP acknowledgments	337
How custom events affect pre-update and validation	337
No events from import tools	338
Generating new messages in Event Fired rules	338
Rule set structure	338
Simple message payload	339
Multiple messages for one event	339
Determining what changed	339
Rule sets must never call message methods for ACK, error, or skip	340
Save values across rule set executions	340
Creating a payload by using Gosu templates	341
Setting a message root object or primary object	341

Creating XML payloads by using GX models	342
Using Java code to generate messages	342
Saving attributes of a message	343
Maximum message size	343
If multiple events fire, which message sends first?	343
Restrictions on entity data in messaging rules and messaging plugins	343
Event fired rule set restrictions for entity data changes	344
Messaging plugin restrictions for entity data changes	344
Messaging interacts with PolicyCenter workflow	345
Database transactions when creating messages	345
Messaging plugins must not call SOAP APIs on the same server	345
Delaying or censoring message generation	346
Validity and rule-based event filtering	346
Late binding data in your payload	346
Implement late binding	347
Tracking a specific entity with a message	348
Implementing message plugins	349
Ensuring thread safety in messaging plugin code	349
Initializing a messaging plugin	349
Getting messaging plugin parameters from the plugin registry	349
Implementing message payload transformations before send	350
Implementing a message transport plugin	351
Implementing post-send processing	352
Implementing a MessageReply plugin	353
Error handling in messaging plugins	355
Suspend, resume, and shut down a messaging destination	356
Map message payloads by using tokens	357
Message status code reference	358
Reporting acknowledgments and errors	360
Message sending error behaviors	360
Submitting ACKs, errors, and duplicates from messaging plugins	361
Using web services to submit ACKs and errors from external systems	362
Using web services to retry messages from external systems	362
Message error handling	363
Resynchronizing messages for a primary object	364
Cloning new messages from pending messages	366
How resynchronization affects preupdate and validation	366
Resync in ContactManager	366
Monitoring messages	366
Messaging tools web service	367
Acknowledging messages	367
Getting the ID of a message	367
Retrying messages	368
Skipping a message	369
Resynchronizing an account for a destination	369
Purging completed messages	370
Suspending a destination	370
Resuming a destination	371
Getting messaging statistics	372
Getting the status of a destination	372
Getting configuration information from a destination	372
Changing messaging destination configuration parameters	373
Included messaging transports	374
Email message transport	374
Register and enable the console message transport	374

Part 5

Policy-related integrations

19 Rating integration	379
The rating framework.....	379
Overview of cost data objects.....	381
Where to override the default rating engine	383
Common questions about the default rating engine.....	385
Optional asynchronous rating.....	388
Implementing rating for a new line of business	389
What do cost data objects contain?	390
Cost core properties	391
Adding line-specific cost properties and methods	395
Fixed ID keys link a cost data object to another object	395
Cost data object methods and constructors to override.....	396
Cost data APIs that you can call	402
Checklist for relationship changes in cost data objects.....	403
Writing your own line-specific rating engine subclass.....	404
A rating line example for personal auto - PersonalAutoCovCostData.....	413
A close look at PersonalAutoCovCostData	413
Constructors for PersonalAutoCovCostData	414
Setting specific properties on cost for PersonalAutoCovCostData.....	414
Versioned costs for PersonalAutoCovCostData	414
Key values for PersonalAutoCovCostData.....	415
Rating slice details for PersonalAutoCovCostData	415
Rate window method for PersonalAutoCovCostData	416
Rating variations	417
Workers' compensation rating	417
Inland marine rating	419
General liability rating	419
20 Guidewire Rating Management and PCRatingPlugin	421
Enable the rating plugin for Guidewire Rating Management	421
Extending Guidewire Rating Management to other lines of business	422
Configuring parallel rating	422
Enabling and configuring parallel rating	422
Implementing parallel rating	423
Extend parallel rating using entities to another line of business	426
Extending parallel rating using DTOs to other lines of business	426
21 Reinsurance integration	431
Reinsurance data model	432
Risk entity	432
Risk version list entity	433
Reinsurance plugin	434
Architecture of the Reinsurance Management plugin	435
Configuration plugins for Guidewire Reinsurance Management	436
Creating a plugin implementation for your own reinsurance system	436
Reinsurance plugin interface methods	437
Reinsurance configuration plugin	439
Reinsurance configuration plugin implementations	440
Reinsurance configuration plugin methods and properties	440
Reinsurance ceding plugin	442
Reinsurance ceding plugin implementations	443
Reinsurance ceding plugin methods	443

Reinsurance coverage web service	445
Methods of the PolicyCenter reinsurance coverage web service	445
22 Forms integration	449
Forms inference classes	449
Creating inference data and XML	450
Determining whether to add or reprint a form	452
Form data helper functions	452
Handling multiple instances of one form	453
Reference dates on a form	455
Forms messaging	455
Messaging plugins	455
Creating documents	455
23 Policy difference and comparison customization	457
Overview of policy differences	457
Difference item subclasses	459
Comparing branches by using difference helper classes	460
Difference APIs	461
Difference methods on the DiffHelper class	461
Difference methods on the DiffUtils class	461
Comparing individual objects by using matchers	462
Planning how to match entity instances	463
Writing delegate-based matcher classes	463
Customizing the classes that perform matching	464
Copier classes	464
Bean matcher classes	465
Important files for customizing policy differences	465
Filtering difference items after creation of database records	466
Difference tree XML configuration	466
Editing the difference tree XML files	469
Customizing differences for new lines of business	474
Filtering difference items	474
Customizing personal auto line of business	475
Methods for calculating differences	476
Differences between a branch and its based-on branch	476
Differences between any two branches	476
24 Claim and policy integration	479
Claim search from PolicyCenter	479
Get claim details	480
Permission to view a claim	480
Policy system notifications	480
Receiving a notification of a large loss	480
PolicyCenter implementation of the large loss notification web service	480
Claim to policy system notification web service API	481
Policy search web service for claim system integration	481
Retrieving a policy	481
Extend the search criteria	482
Typecode maximum length and trimmed typecodes	482
Policy search SOAP API	482
PolicyCenter exit points to ClaimCenter and BillingCenter	483

Part 6

Claim and policy integrations

24 Claim and policy integration	479
Claim search from PolicyCenter	479
Get claim details	480
Permission to view a claim	480
Policy system notifications	480
Receiving a notification of a large loss	480
PolicyCenter implementation of the large loss notification web service	480
Claim to policy system notification web service API	481
Policy search web service for claim system integration	481
Retrieving a policy	481
Extend the search criteria	482
Typecode maximum length and trimmed typecodes	482
Policy search SOAP API	482
PolicyCenter exit points to ClaimCenter and BillingCenter	483

PolicyCenter Product Model import into ClaimCenter	483
Configuring the ClaimCenter Typelist Generator	485
Run the ClaimCenter Typelist Generator	489
Using generated typelists in ClaimCenter	490
Typelist localization	495
Policy location search API	496
Dependencies of the policy location search API	496
Finding policy locations within geographic bounding boxes	496

Part 7**Billing integrations**

25 Billing integration	501
Billing integration overview	501
Mechanisms for integrating PolicyCenter and a billing system	502
Integrating PolicyCenter and BillingCenter	502
Integrating policy center with another billing system	503
How billing data flows between applications	503
Notifying a billing system of policy changes and premiums	503
Tracking policies term by term	504
PolicyCenter properties for billing system usage	504
Asynchronous communication between PolicyCenter and billing system	504
Exit points between PolicyCenter and billing system applications	506
Configuring which system receives contact updates	507
Using integration-specific containers for integration	507
Billing producers and producer codes	507
Using producer information in BillingCenter	507
Comparison of producer codes in PolicyCenter and BillingCenter	508
PolicyCenter producer to BillingCenter properties mapping	508
Billing accounts	509
PolicyCenter billing account to BillingCenter properties mapping	509
Billing plan	511
Delinquency plan	511
Invoicing	511
Contacts	512
Policy period merges	512
Service tier	512
Billing instructions in BillingCenter	513
Billing instruction subtypes	513
PolicyCenter financials to BillingCenter charge properties mapping	515
Sending PolicyCenter transaction information to BillingCenter	516
Billing flow for new-period jobs	516
Flow of submission, renewal, and rewrite	516
Policy period mapping	518
Billing methods and payment plans	519
New periods and term confirmed flag	519
Billing flow for existing-period jobs	519
Billing implications of midterm changes	520
Midterm changes to a policy	520
Midterm changes to billing method or payment plan	520
Holding billing on midterm policy transaction charges	520
Midterm changes to producer of record or producer of service	521
Moving a policy to a new account in midterm	521
Billing implications of renewals or rewrites	521

Choosing the renewal flow type	521
Account creation for conversion on renewal	526
Copying billing data to new periods on renewal or rewrite	526
Billing implications for cancellations and reinstatements	526
Cancellations that start in PolicyCenter	527
Cancellations that start in BillingCenter	527
Billing implications of audits	528
Holding periods open for audits	528
Generating an audit report	528
Sending audit premiums as incremental	529
Audit reversals and revisions	529
Billing implications for premium reporting	530
Billing implications of delinquency for failure to report	530
Billing implications of deposits	531
Deposits in submission, renewal, or rewrite jobs	531
Deposits in policy change and reinstatement jobs	532
Deposits in premium reports	532
Deposits in cancellation	532
Deposits in final audit	533
Billing implications of up-front payments	533
Implementing the billing system plugin	533
Account management	534
Policy period management	535
Getting period information from the billing system	536
Billing system notifications of PolicyCenter policy actions	536
Getting the billing level	539
Producer management	540
Agency bill plan availability retrieval	541
Commission plan management	541
Payment plans and installment previews	542
Updating contacts	544
Other billing system plugin methods	545
Implementing the billing summary plugin	545
Interfaces used by the billing summary plugin	545
Getting policies billed to accounts	546
Retrieving account billing summaries	546
Retrieving account invoices	546
Retrieving billing summaries for policy periods	546
Retrieving policy billing summary	546
Payment integration	547
Overview of integration with a payment system	547
Entities and classes for tracking payments	547
Integration points to external payment systems	548
Accessing an external payment system to add a new payment instrument	548
Configure PolicyCenter to use a real payment system	549
Accessing an external payment gateway to add an up-front payment	550
Configuring PolicyCenter to use a real payment gateway	552
26 Contact integration	565
Integrating with a contact management system	565
Inbound contact integrations	566

Part 8

Contact integrations

26 Contact integration	565
Integrating with a contact management system	565
Inbound contact integrations	566

Asynchronous messaging with the contact management system	566
Retrieve a contact	567
Contact searching	568
Support for finding duplicate contacts	569
Finding duplicate contacts	569
Find duplicate contacts	570
Adding contacts to the external system	570
Updating contacts in the external system	570
Overwriting the local contact with latest values	571
Configuring how PolicyCenter handles contacts	571
Configuring available account contact role types	571
Configuring mapping of contact type to account contact role	571
Configuring account contact role type display name	571
Configuring account contact role type for a policy contact role	572
Configuring allowed contact types for policy contact role type	572
Configuring whether to treat an account contact type as available	572
Synchronizing contacts with accounts	572
Account contact plugin	573
Account contact role plugin	573
Contact web service APIs	573
Deleting a contact	574
Adding a contact	575
Updating a contact	575
Merging contact addresses	575
Merging contacts	576
Handling rejection and approval of pending changes	577
Activating and deactivating contacts	577
Getting associated policy transactions for a contact	578
Getting associated policies for a contact	578
Getting associated accounts for a contact	578
Address APIs	579
Updating an address	579

Part 9

Importing policy data

27 Importing from database staging tables	583
Importing zone data	583
High-level steps to import zone data	584
Files for zone data import	584
Zone data files supplied by Guidewire	584
Zone data configuration files	585
Database import tables and columns	585
Staging tables	585
Load error table	586
Load history table	586
Load command IDs	586
Load user IDs	587
Load time stamps	587
Import tools and commands	587
Server modes and run levels for database staging table import	587
Database consistency checks	588
Integrity checks	588
Database performance considerations	589

Loading zone data into staging tables	590
Clearing errors from staging tables	590
Importing data into operational tables	590
Detailed work flow for a typical database staging table import	591
Prepare the data and the PolicyCenter database	592
Import data into the staging tables	592
Load staging table data into operational tables	593
Perform post-import tasks	593
Table import tips and troubleshooting	594

Part 10

Other integration topics

28 Guidewire InsurancePlatform Integration Views.	597
Overview of Integration Views	597
Creating an Integration View	598
Mapping a data object to an Integration View schema	598
Creating a schema for Integration View objects	599
Filtering Integration View objects for select attributes	601
Accessing and naming files related to Integration Views in Guidewire Studio	601
Handling an Integration View	601
Handling an Integration View using a REST API	602
Handling an Integration View using an event message handler	602
29 GX models.	605
Create a GX model	605
Including a GX model inside another GX model	607
Mappable and unmappable properties	607
Normal and key properties	608
Automatic publishing of the generated XSD schema	608
Create an instance of a GX model	608
GXOptions	609
GX model labels	610
Assign a label to a GX model property	610
Reference a GX model label	610
Create an instance of a GX model with labels	611
GX model label example	612
Serialize a GX model object to XML	613
Arrays of entities in XML output	613
Sending a message only if data model fields changed	613
Conversions from Gosu types to XSD types	614
30 Archiving integration.	617
Overview of archiving integration	618
Basic integration flow for archiving storage	618
Basic integration flow for archiving retrieval	618
PolicyCenter web service for archiving	619
Check for archiving before accessing policy data	619
Upgrading the data model of retrieved data	619
Error handling during archive	620
Archiving storage integration detailed flow	620
Archiving retrieval integration detailed flow	621
Archive source plugin	622
Archive source plugin methods and archive transactions	623
Archive source plugin storage methods	624

Archive source plugin retrieval methods	626
Archive source plugin utility methods	627
Archiving eligibility plugin.	629
31 Custom processing	631
Overview of custom processes	631
Processing modes	631
Choosing a mode for a custom process	632
About scheduling PolicyCenter processes	632
About manually executing PolicyCenter processes	633
Batch processing typecodes	633
Custom work queues	633
About custom work queue classes	634
Work queues and work item entity types	634
Work queues that use StandardWorkItem	635
Lifecycle of a work item	635
Considerations for developing a custom work queue	636
Define a typecode for a custom work queue	638
Define a custom work item type	638
Creating a custom work queue class	639
Developing the writer for your custom work queue	640
Developing workers for your custom work queue	641
Bulk insert work queues	643
Overview of bulk insert work queues	643
Bulk insert work queue declaration and constructor	643
Querying for targets of a bulk insert work queue	644
Eliminating duplicate items in bulk insert work queue queries	644
Processing work items in a custom bulk insert work queue	644
Example work queue that bulk inserts its work items	645
Examples of custom work queues	646
Example work queue that extends WorkQueueBase	646
Example work queue for updating entities	647
Example work queue with a custom work item type	648
Developing custom batch process	650
Custom batch process overview	650
Create a custom batch process	651
Define a typecode for a custom batch process	651
Batch process base class	652
Useful properties on class BatchProcessBase	652
Useful methods on class BatchProcessBase	653
Examples of custom batch processes	657
Example batch process for a background task	657
Example batch process for unit of work processing	658
Working with custom processing	660
Categorizing a process typecode	660
Updating the work queue configuration	661
Implementing IProcessesPlugin	661
Monitoring PolicyCenter processes	663
Monitoring PolicyCenter processing using administrative screens	663
Monitoring PolicyCenter processes for completion	664
Monitoring batch processes by using maintenance tools	664
Periodically purging process entities	664
Managing user entity updates in batch processing	665
Set the external user field in a batch process	665

32 Free-text search integration	667
Free-text search plugins overview	667
Connecting the free-text plugins to the Guidewire Solr Extension.	668
Enabling and disabling the free-text plugins.	668
Running the free-text plugins in debug mode	668
Free-text load and index plugin and message transport.	668
Message destination for free-text search	669
Enable the message destination for free-text search.	669
ISolrMessageTransportPlugin plugin parameters	670
Free-text search plugin	670
ISolrSearchPlugin plugin parameters	670
33 Servlets	673
Implementing servlets	673
@Servlet annotation	674
Servlet definition in servlets.xml	675
Important HTTPServletRequest object properties	676
Create and test a basic servlet.	676
Example of a basic servlet	678
Implementing servlet authentication.	678
Create a servlet that provides basic authentication	679
Test your basic authentication servlet	679
Example of a basic authentication servlet	680
Supporting multiple authentication types.	681
Abstract HTTP basic authentication servlet class	681
Abstract Guidewire authentication servlet class	681
ServletUtils authentication methods	681
Example of a servlet using multiple authentication types.	683
Test your servlet using multiple authentication types.	683
34 Database connection pool.	685
Reserving a single database connection	685
35 Data extraction integration.	687
Data extraction using web services	688
Using Gosu templates for data extraction.	689
Gosu template APIs for data extraction integration	690
36 Proxy servers.	691
Configuring a proxy server with Apache HTTP Server.	692
Install Apache HTTP Server: Basic checklist.	692
Certificates, private keys, and passphrase scripts	692
Proxy server integration types for PolicyCenter.	693
Bing maps geocoding service communication	693
Proxy building blocks	694
Downstream proxy with no encryption	694
Downstream proxy with encryption	694
Upstream (reverse) proxy with encryption for service connections	695
Upstream (reverse) proxy with encryption for user connections	696
Modify the server to receive incoming SSL requests.	696
37 Java and OSGi support	699
Implementing plugin interfaces in Java and optionally OSGi	699
Choosing between a Java plugin and an OSGi plugin	700
Your IDE options for plugin development in Java.	700
Java IDE inspections that flag unsupported internal APIs	701

Install the internal API inspection in IntelliJ IDEA	701
Accessing entity data from Java	702
Regenerate Java API libraries	702
Java API reference documentation	703
Accessing entity instances from Java	703
Accessing typecodes from Java	703
Getting a reference to an existing bundle from Java	704
Creating a new bundle from Java	704
Creating a new entity instance from Java	704
Querying for entity data from Java	705
Using reflection to access Gosu classes from Java	705
Using Gosu enhancement properties and methods from Java	707
Class loading and delegation for non-OSGi Java	707
Deploying non-OSGi Java classes and JAR files	708
Deploy an example Java class with no plugins and no entities	708
Using IntelliJ IDEA with OSGi editor to deploy an OSGi plugin	709
Create a new project with an OSGi plugin module	709
Create an OSGi-compliant class that implements a plugin interface	711
Example startable plugin in Java using OSGi	711
The OSGi Ant build script	712
Compile and install your OSGi plugin as an OSGi bundle	712
Configuring third-party libraries in an OSGi plugin	714
Advanced OSGi dependency and settings configuration	717
OSGi dependencies	717
OSGi build properties	717
OSGi bundle metadata configuration file	718
OSGi build configuration Ant script	718
Update your OSGi plugin project after product location changes	718
38 Inbound files integration	721
Work queues	721
InboundFilePurgeWorkQueue	722
InboundChunkWorkQueue	722
Implementing an integration	722
Create an InboundFileHandler implementation	722
Configuring inbound files integration	724
39 Outbound files integration	729
Work queues	729
OutboundFilePurgeWorkQueue	730
OutboundRecordPurgeWorkQueue	730
Implementing an integration	730
Create an OutboundFileHandler implementation	730
Configure outbound files integration	731
Part 11	
Reference specifications	
40 Integration mapping specification	737
Integration mapping overview	737
Integration mapping files	737
Integration mapping file specification	738
Integration mapping file combination	740
Integration mapping file imports	741
Integration mappers	742

Integration View filters	744
JsonMapper and TransformResult	746
Integration mapping examples	747
41 Guidewire JSON schema support specification	751
JSON schema files	751
JSON schema file specification	752
JSON schema file combination	761
JSON schema imports	762
JSON data types and formats	764
JSON parsing and validation	766
JSON serialization	768
JsonObject class	769
JSON schema wrapper classes	770
XML output and XSD translation	772
Releasing schemas	776
Externalized JSON schemas	777

Guidewire Documentation

Guidewire InsurancePlatform™

Guidewire InsurancePlatform™ is the property and casualty industry platform that unifies software, services, and a partner ecosystem to power our customers' businesses. InsurancePlatform provides the standard upon which insurers can optimize their operations, increase engagement, drive smart decisions, innovate quickly, and simplify IT.

This documentation provides information on installation, implementation, administration, and customization for Guidewire InsurancePlatform™ software and SaaS offerings.

About PolicyCenter documentation

The following table lists the documents in PolicyCenter documentation:

Document	Purpose
<i>InsuranceSuite Guide</i>	If you are new to Guidewire InsuranceSuite applications, read the <i>InsuranceSuite Guide</i> for information on the architecture of Guidewire InsuranceSuite and application integrations. The intended readers are everyone who works with Guidewire applications.
<i>Application Guide</i>	If you are new to PolicyCenter or want to understand a feature, read the <i>Application Guide</i> . This guide describes features from a business perspective and provides links to other books as needed. The intended readers are everyone who works with PolicyCenter.
<i>Upgrade Guide</i>	Describes the overall upgrade process, and describes how to upgrade your configuration and database. The intended readers are system administrators and implementation engineers who must merge base application changes into existing application extensions and integrations. Visit the Guidewire Community to access the <i>Upgrade Guide</i> , which is available for download, separately from the main documentation set, with the Guidewire InsuranceSuite Upgrade Tools.
<i>Installation Guide</i>	Describes how to install PolicyCenter. The intended readers are everyone who installs the application for development or for production.
<i>System Administration Guide</i>	Describes how to manage a PolicyCenter system. The intended readers are system administrators responsible for managing security, backups, logging, importing user data, or application monitoring.
<i>Configuration Guide</i>	The primary reference for configuring initial implementation, data model extensions, and user interface (PCF) files for PolicyCenter. The intended readers are all IT staff and configuration engineers.
<i>PCF Format Reference</i>	Describes PolicyCenter PCF widgets and attributes. The intended readers are configuration engineers. See the <i>Configuration Guide</i>
<i>Data Dictionary</i>	Describes the PolicyCenter data model, including configuration extensions. The dictionary can be generated at any time to reflect the current PolicyCenter configuration. The intended readers are configuration engineers.
<i>Security Dictionary</i>	Describes all security permissions, roles, and the relationships among them. The dictionary can be generated at any time to reflect the current PolicyCenter configuration. The intended readers are configuration engineers.
<i>Globalization Guide</i>	Describes how to configure PolicyCenter for a global environment. Covers globalization topics such as global regions, languages, date and number formats, names, currencies, addresses, and phone numbers. The intended readers are configuration engineers who localize PolicyCenter.
<i>Rules Guide</i>	Describes business rule methodology and the rule sets in Guidewire Studio for PolicyCenter. The intended readers are business analysts who define business processes, as well as programmers who write business rules in Gosu.

Document	Purpose
<i>Guidewire Contact Management Guide</i>	Describes how to configure Guidewire InsuranceSuite applications to integrate with ContactManager and how to manage client and vendor contacts in a single system of record. The intended readers are PolicyCenter implementation engineers and ContactManager administrators.
<i>Best Practices Guide</i>	A reference of recommended design patterns for data model extensions, user interface, business rules, and Gosu programming. The intended readers are configuration engineers.
<i>Integration Guide</i>	Describes the integration architecture, concepts, and procedures for integrating PolicyCenter with external systems and extending application behavior with custom programming code. The intended readers are system architects and the integration programmers who write web services code or plugin code in Gosu or Java.
<i>Java API Reference</i>	Javadoc-style reference of PolicyCenter Java plugin interfaces, entity fields, and other utility classes. The intended readers are system architects and integration programmers.
<i>Gosu Reference Guide</i>	Describes the Gosu programming language. The intended readers are anyone who uses the Gosu language, including for rules and PCF configuration.
<i>Gosu API Reference</i>	Javadoc-style reference of PolicyCenter Gosu classes and properties. The reference can be generated at any time to reflect the current PolicyCenter configuration. The intended readers are configuration engineers, system architects, and integration programmers.
<i>Testing Guide</i>	Describes the tools and functionality provided by InsuranceSuite for testing application behavior during an initial implementation or an upgrade. The guide covers functionality related to Behavior Testing Framework, GUnit, and Gosu functionality designed specifically for application testing. There are two sets of intended readers: business analysts who will assist in writing tests that describe the desired application behavior; and technical developers who will write implementation code that executes the tests.
<i>Glossary</i>	Defines industry terminology and technical terms in Guidewire documentation. The intended readers are everyone who works with Guidewire applications.
<i>Product Model Guide</i>	Describes the PolicyCenter product model. The intended readers are business analysts and implementation engineers who use PolicyCenter or Product Designer. To customize the product model, see the <i>Product Designer Guide</i> .
<i>Product Designer Guide</i>	Describes how to use Product Designer to configure lines of business. The intended readers are business analysts and implementation engineers who customize the product model and design new lines of business.
<i>REST API Framework</i>	Describes the Guidewire InsuranceSuite framework that provides the means to define, implement, and publish REST API contracts. It also describes how the Guidewire REST framework interacts with JSON and Swagger objects. The intended readers are system architects and integration programmers who write web services code or plugin code in Gosu or Java.

Conventions in this document

Text style	Meaning	Examples
<i>italic</i>	Indicates a term that is being defined, added emphasis, and book titles. In monospace text, italics indicate a variable to be replaced.	<p><i>A destination</i> sends messages to an external system.</p> <p>Navigate to the PolicyCenter installation directory by running the following command:</p> <pre>cd installDir</pre>
bold	Highlights important sections of code in examples.	<pre>for (i=0, i<someArray.length(), i++) { newArray[i] = someArray[i].getName() }</pre>
narrow bold	The name of a user interface element, such as a button name, a menu item name, or a tab name.	Click Submit .

Text style	Meaning	Examples
<code>monospace</code>	Code examples, computer output, class and method names, URLs, parameter names, string literals, and other objects that might appear in programming code.	The <code>getName</code> method of the <code>IDoStuff</code> API returns the name of the object.
<code>monospace italic</code>	Variable placeholder text within code examples, command examples, file paths, and URLs.	Run the <code>startServer server_name</code> command. Navigate to http://server_name/index.html .

Support

For assistance, visit the Guidewire Community.

Guidewire customers

<https://community.guidewire.com>

Guidewire partners

<https://partner.guidewire.com>

Part 1

Planning integration projects

Overview of integration methods

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

PolicyCenter 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 policy validation logic, use a legacy system that generates policy numbers, or query a legacy system.

Sending messages to external systems in a transaction-safe way

Trigger actions after important events happen in PolicyCenter, and notify external systems only if the change is successful and no exceptions occurred. For example, alert a policy database if anyone changes policy 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 PolicyCenter. You can choose which methods make the most sense for your integration project.

Using clearly defined industry standard protocols for integration points

PolicyCenter provides APIs to retrieve policies, 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 PolicyCenter integration framework provides multiple ways to integrate external code with PolicyCenter.

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 programmatically add submissions, policy revisions, and policies to PolicyCenter.

PolicyCenter 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

PolicyCenter plugins are classes that PolicyCenter 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. PolicyCenter 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 PolicyCenter 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. PolicyCenter 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 policy numbers (`IPolicyNumGenAdapter`) or get a claim related to a policy from a claims system (`IClaimSearchAdapter`).

Templates

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

The following table compares the main integration methods.

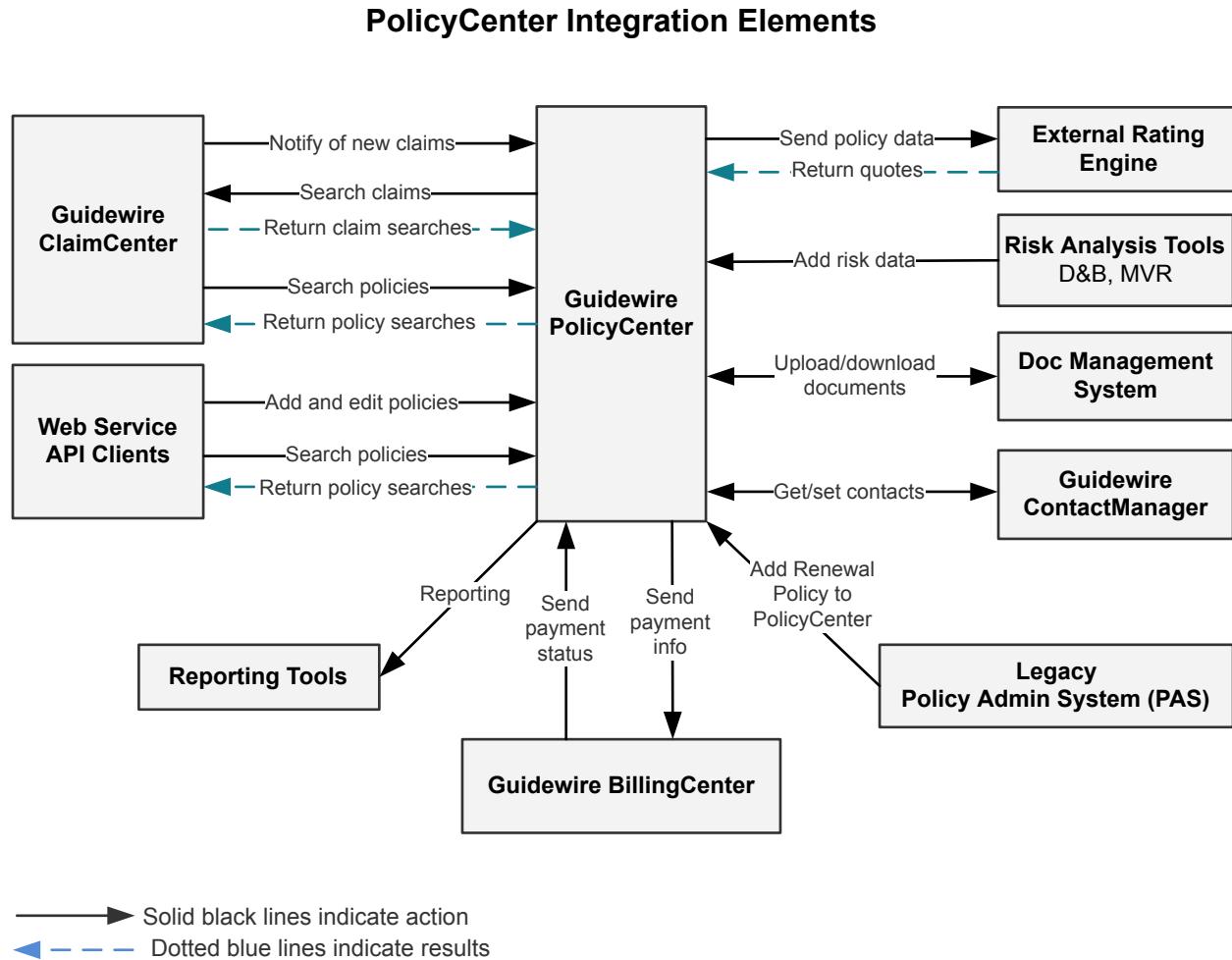
Integration type	Description	What you write
Web services	A full API to manipulate PolicyCenter data and trigger actions externally from any programming language or platform that uses PolicyCenter Web services APIs. Accessed by using the platform-independent SOAP protocol.	<ul style="list-style-type: 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 PolicyCenter generates. To consume external web services from Gosu, write Gosu code that uses WSDL from the external system. Gosu creates native types

Integration type	Description	What you write
		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 PolicyCenter. The general name for such objects are Data Transfer Objects (DTOs). Define DTOs as Gosu classes or by using XSDs.	
Plugins	Classes that PolicyCenter calls to perform tasks or calculate a result. Plugins run in the same Java virtual machine (JVM) as PolicyCenter.	• 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.	• 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 send the messages to external systems. The most important interface is MessageTransport. 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 PolicyCenter. The general name for such objects is Data Transfer Objects (DTOs). Define DTOs as Gosu classes or by using XSDs.
GX models	<p>GX models enable the properties of an entity or other data type to be converted to XML. The GX model editor in Studio defines the data type's properties to include in the GX model. As the model is defined, an XSD schema definition is automatically generated.</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>	• Using the GX Model editor in Studio, you customize the properties in an entity or other data type to export in XML format. • Various integration code that uses the GX model.
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"> • Send notifications as form letters and use plain text with embedded Gosu code to simplify deployment and ongoing updates. • Design a template that exports HTML for easy development of web-based reports of PolicyCenter data. 	• Text files that contain small amounts of Gosu code.

Integration type	Description	What you write
Links	Link from a PolicyCenter 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"> Create an ExitPoint PCF file. Incorporate the ExitPoint into the originating PolicyCenter screen.

PolicyCenter integration elements

The following diagram summarizes some common elements of policy management using PolicyCenter.



See also

- For information about integrating with BillingCenter, see “Billing integration” on page 501.
- For information about integrating with ClaimCenter, see “Claim and policy integration” on page 479.
- For information about integrating with ContactManager, see the *Guidewire Contact Management Guide*.

Preparing for integration development

During integration development, edit configuration files in the hierarchy of files in the product installation directory. In most cases, you modify data only through the 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 certain 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:

```
PolicyCenter/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/web	Your web application configuration files, also known as PCF files.
config/logging	The logging configuration file log4j2.xml.
config/templates	<p>Two types of templates relevant for integration.</p> <p>Plugin templates</p> <p>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 PolicyCenter needs to call the plugin, PolicyCenter passes the template results to the plugin as String parameters.</p> <p>Messaging templates</p> <p>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.</p> <p>The base configuration provides versions of some of these templates.</p>
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, PolicyCenter 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 PolicyCenter installation directory	Files that the directory contains
PolicyCenter 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 PolicyCenter server. Almost all of these tools are small Gosu scripts that call public web service APIs.

Overview of integration documentation for programmers

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 PolicyCenter 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 51.

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

The *PolicyCenter 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.

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 PolicyCenter release.
- After you make changes to the PolicyCenter 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 PolicyCenter 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:

```
PolicyCenter/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:

```
PolicyCenter/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 PolicyCenter 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 PolicyCenter to calculate a value or perform a task.

Java API libraries

Entities, which are PolicyCenter types defined in the data model with built-in properties and can also have data model extension properties. For example, **Policy** and **Address** are PolicyCenter entities. To access entity data and methods from Java, you need to use Java API libraries.

In all cases, PolicyCenter entities such as **Policy** contain data properties that can be manipulated either directly or from some contexts 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 arguments or return values.	n/a

Entity access	Description	Entities	Necessary libraries
		Instead, create your own data transfer objects (DTOs) as one of the following:	<ul style="list-style-type: none"> • Gosu class instances that contain only the properties required for each integration point. You also need to declare the class <code>final</code> and add the annotation <code>gw.xml.ws.annotation.WsiExportable</code>. • XML objects with structure defined in XSD files. • XML objects with 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

PolicyCenter creates its own unique ID for every entity in the system after it fully loads in the PolicyCenter 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 PolicyCenter 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 policy and a policy revision may share the same public ID because they are different entities.

If loading two related objects, the incoming request must tell PolicyCenter that they are related. However, the web service client does not know the internal PolicyCenter 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 PolicyCenter about changes to an object even though the external system might not know the internal ID that PolicyCenter 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.

PolicyCenter 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 PolicyCenter 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 PolicyCenter 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 PolicyCenter 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`, PolicyCenter generates a public ID. The ID is a two-character ID, followed by a colon, followed by a server-created number. For example, "`pc: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 PolicyCenter-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 PolicyCenter must never use an internal class nor call any method of an internal class for any reason. Future releases of PolicyCenter 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.

Part 2

Web services

Overview of 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 PolicyCenter), Java, Perl, and other languages.

Remote systems call PolicyCenter 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.

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

PolicyCenter web services are written in the Gosu programming language. PolicyCenter 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, PolicyCenter 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 PolicyCenter, 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, PolicyCenter 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, PolicyCenter 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 PolicyCenter to calculate values, to trigger actions, or to change data in the PolicyCenter 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 base configuration web services

The following table lists all base configuration web services.

Web Service Name	Description
Job-related web services	
CancellationAPI	Starts a cancellation job or rescinds a cancellation that has not completed. See "Policy cancellation and reinstatement web services" on page 116.
JobAPI	Adds an activity to a job. See "Job web services" on page 116.
PolicyChangeAPI	Starts a policy change job. See "Policy change web services" on page 125.
PolicyRenewalAPI	Starts a renewal job. See "Policy renewal web services" on page 126.
ReinstatementAPI	Starts a reinstatement job for a policy that has been canceled. See "Policy cancellation and reinstatement web services" on page 116.
SubmissionAPI	Starts a submission job. See "Submission web services" on page 119.
Other application web services	
AccountAPI	Performs various actions on an account, such as adding a note or a document. See "Account web services" on page 114.

Web Service Name	Description
AddressAPI	Notifies PolicyCenter of updates to addresses from an external contact system. See “Address APIs” on page 579.
ArchiveAPI	Manipulates archiving from external systems. See “Archiving web services” on page 128.
CCPolicySearchIntegration	Retrieves policy summaries for claim systems such as ClaimCenter. If you configure ClaimCenter to connect to PolicyCenter, ClaimCenter uses this web service automatically. See “Policy search web service for claim system integration” on page 481.
ClaimToPolicySystemNotificationAPI	Notifies PolicyCenter about large losses on a claim. If you configure ClaimCenter to connect to PolicyCenter, ClaimCenter uses this web service automatically. See “Policy system notifications” on page 480.
ContactAPI	Notifies PolicyCenter of changes to contacts, including merging of contacts and merging of contact addresses. See “Contact web service APIs” on page 573.
ImportPolicyAPI	Adds an activity to a policy. See “Import policy web services” on page 125.
PolicyAPI	Adds an activity to a policy. See “Policy web services” on page 122.
PolicyEarnedPremiumAPI	Calculates earned premiums for a policy number as of a specific date. See “Policy earned premium web services” on page 124.
ProductModelAPI	Modifies the product model on a running development (non-production) server. See “Product Model web services” on page 120.
PolicyLocationSearchAPI	Retrieves a list of policy location summaries within a rectangular geographic bounding box. See “Policy location search API” on page 496.
PolicyPeriodAPI	Performs various actions on a policy period, such as adding a note or a document. See “Policy period web services” on page 124.
PolicySearchAPI	Retrieves the public ID of a policy period based on a policy number and a date. See “Policy search web service for claim system integration” on page 481.
ProducerAPI	Performs various actions on producers, agencies, and branches. See “Producer web services” on page 120.
ProductModelAPI	Modifies the Product Model on a development (non-production) server. See “Product Model web services” on page 120.
RICoverageAPI	Finds reinsurance risk information for use by an external system. This web service requires <i>Guidewire Reinsurance Management</i> , which you license separately from PolicyCenter. See “Reinsurance coverage web service” on page 445.
General web services	
AdminDataAPI	For Guidewire use only. Do not use.
DataChangeAPI	Tool for rare cases of mission-critical data updates on running production systems. See the <i>System Administration Guide</i> for more information.
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. See “Importing administrative data” on page 95.
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. For details, see “Login authentication confirmation” on page 59.
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. See “Maintenance tools web service” on page 96.

Web Service Name	Description
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. See “Messaging tools web service” on page 367.
ProfilerAPI	Sends information to the built-in system profiler. See “The Profiler API web service” on page 102.
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. See “System tools web service” on page 104.
TemplateToolsAPI	Lists and validates Gosu templates available on the server. See “Template web service” on page 109.
TableImportAPI	Loads geographic zone data from the staging table into the operational table. In PolicyCenter, this tool supports zone data only, not other data such as policy data or administrative data. Key methods of this web service are available only when the server run level is maintenance. See “Table import web service” on page 107.
TypeListToolsAPI	Retrieves aliases for PolicyCenter typecodes in external systems. See “Mapping typecodes and external system codes” on page 99.
WorkflowAPI	Performs various actions on a workflow, such as suspending and resuming workflows and invoking workflow triggers. See “Workflow web service” on page 110.
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. See “Zone data import web service” on page 111.

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, PolicyCenter publishes web service APIs for use by external programs and Guidewire applications.

Web service publishing quick reference

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

Feature	Web service behavior
Basic publishing of a web service	Add the annotation <code>@WsWebService</code> 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. See “Declaring the namespace for a web service” on page 47. If you do not declare the namespace, Gosu uses the default namespace http://example.com .
Does PolicyCenter automatically generate WSDL files from a running server?	Yes. See “Generating and publishing WSDL” on page 51.
Does PolicyCenter automatically generate WSDL files locally if you regenerate the SOAP API files?	Yes. See “Generating and publishing WSDL” on page 51.
Does PolicyCenter automatically generate JAR files for Java SOAP client code if you regenerate the SOAP API files?	No, but it is easy to generate with the Java built-in utility <code>wsimport</code> . The documentation includes examples. See “Calling a PolicyCenter web service from Java” on page 53.
Can PolicyCenter serialize Gosu class instances, sometimes called POGOs: Plain Old Gosu Objects?	Yes, but with certain requirements. See “Data transfer objects” on page 46.
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 46.

Feature	Web service behavior
Can PolicyCenter 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 <code>XmlElement</code> APIs. See “Using predefined XSD and WSDL” on page 63.
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?	Generally speaking, no. Exceptions include Simple Java objects. Such objects include but are not limited to <code>Date</code> , <code>Double</code> , and <code>String</code> .
Can web services serialize an enumeration—typelist, Java enum, or Gosu enum—as an argument type or return type?	Yes. See “Exposing typelists and enums as enumeration values and string values” on page 76
Can web services automatically throw <code>SOAPException</code> 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 <code>@Throws</code> annotation.
	<pre data-bbox="589 920 1334 1058"> class MyClass{ @Throws(SOAPServerException, "If communication error or any other SOAP problem occurs.") public function myMethod() { ... } } </pre>
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 47.
Calling a local version of the web service from Gosu.	<p>Gosu creates types that use the original package name to avoid namespace collisions. API references in the package <code>wsi.local.ORIGINAL_PACKAGE_NAME</code></p> <p>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.</p> <p>For testing purposes only, the local files must be rebuilt by running the following command from the PolicyCenter installation directory.</p> <pre data-bbox="589 1607 763 1632">gwb genWsiLocal</pre>
Create SOAP 1.2 custom fault subcodes.	<p>Custom fault subcodes are not supported in PolicyCenter. However, configuration code can throw a <code>gw.xml.ws.WsdlFault</code> exception, which includes properties that can be populated and subsequently retrieved.</p>
	<pre data-bbox="589 1765 1165 1949"> setActorRole(String actorRole) // SOAP 1.1 actor role getActorRole() : String setCode(FaultCode code) getCode() : FaultCode setCodeQName(QName codeQName) getCodeQName() : QName </pre>

Feature	Web service behavior
	<pre>setDetail(XmlElement detail) getDetail() : XmlElement</pre>
The properties returned by the WsdlFault will not be documented in the generated WSDL.	

Web service publishing annotation reference

The following table lists annotations related to web service declaration and configuration.

Annotation	Description
@WsiAdditionalSchemas	Expose 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. “Referencing additional schemas in your published WSDL” on page 66
@WsiAvailability	Set the minimum server run level for this service. “Specifying the minimum run level for a web service” on page 48
@WsiCheckDuplicateExternalTransaction	Detect duplicate operations from external systems that change data. “Checking for duplicate external transaction IDs” on page 75
@WsiExportable	Add this annotation on a Gosu class to indicate that it supports serialization with web services. The annotation accepts an optional String argument that specifies a namespace. “Data transfer objects” on page 46
@WsiExposeEnumAsString	Instead of exposing typelist types and enumerations as enumerations in the WSDL, you can expose them as String values. “Exposing typelists and enums as enumeration values and string values” on page 76
@WsiInvocationHandler	Perform 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. “Using predefined XSD and WSDL” on page 63
@WsiParseOptions	Add validation of incoming requests using additional schemas in addition to the automatically generated schemas. “Validating requests using additional schemas as parse options” on page 67
@WsiPermissions	Set the required permissions for this service. “Specifying required permissions for a web service” on page 48
@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. It can result in connections becoming effectively locked, reducing the efficiency of the connection pool and slowing web service execution. The annotation can be used in special situations only. An example is when a single web service performs a very large number of database queries, where retrieving and returning a pool connection for each query would cause excessive thrashing.

Annotation	Description
@WsiRequestTransform @WsiResponseTransform	Add transformations of incoming or outgoing data as a byte stream. Typically you would use this annotation to add advanced layers of authentication or encryption. Contrast to the annotations in the next row, which operate on XML elements. “Transformations on data streams” on page 70
@WsiRequestXmlTransform @WsiResponseXmlTransform	Add 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 annotations in the previous row, which operate on a byte stream. “Request or response XML structural transformations” on page 60
@WsiSchemaTransform	Transform the generated schema that PolicyCenter publishes. “Transforming a generated schema” on page 60
@WsiSerializationOptions	Add serialization options for web service responses, for example supporting encodings other than UTF-8. “Setting response serialization options, including encodings” on page 70
@WsiWebMethod	<p>The @WsiWebMethod annotation can perform two actions.</p> <ul style="list-style-type: none"> Override the web service operation name to something other than the default, which is the method name. Suppress a method from generating a web service operation even though the method has public access. <p>“Overriding a web service method name or visibility” on page 49</p>
@WsiWebService	<p>Declare a class as implementing a web service.</p> <p>“Publishing and configuring a web service” on page 47</p>

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 example.wsi.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.

PolicyCenter 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, PolicyCenter 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 it.

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 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 is NODAEMONS.

```
package 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 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 web service method names

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 public web service methods

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 method names 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

Property	Description	Writable
MtomEnabled	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. If <code>MtomEnabled</code> is <code>true</code> , PolicyCenter can send MTOM in results. Otherwise, MTOM is disabled. The default is <code>false</code> . This property does not affect MTOM data sent to a published web service. Incoming MTOM data is always supported. 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 93.	Yes
RequestHttpHeaders	Request headers of type <code>HttpHeaders</code> .	No
RequestEnvelope	Request envelope of type <code>XmlElement</code> .	No
RequestSoapHeaders	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 <code>HttpHeaders</code> .	The property value is read-only, but you can modify the object it references.
ResponseSoapHeaders	Response SOAP headers of type <code>List<XmlElement></code> .	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 writable. The property is readable.

Property	Description	Writable
<code>XmlSerializationOptions</code>	XML serialization options of type <code>XmlSerializationOptions</code> .	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

PolicyCenter does not instantiate a web service implementation class on server startup. PolicyCenter 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 PolicyCenter 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.

PolicyCenter 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 may make references to schemas at the following URL.

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

For example, PolicyCenter generates and publishes the WSDL for web service class `example.webservice.TestWsiService` at the location on a server with web application name pc.

```
http://localhost:8180/pc/ws/example/webservice/TestWsiService?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:8180/pc/ws
```

From there, you can browse 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!"
    }
}
```

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

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

PolicyCenter 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:xs="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="xs:string" minOccurs="0"/>
    </xss:sequence>
</xss:complexType>
</xss:element>
</xss:schema>
</wsdl:types>
<wsdl:portType name="HelloWorldAPIPortType">
    <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="HelloWorldAPIPortType">
    <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="HelloWorldAPIPortType">
    <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/pc/ws/example/HelloWorldAPI"/>
        <gw:address location="${pc}/ws/example/HelloWorldAPI"/>
    </wsdl:port>
    <wsdl:port name="HelloWorldAPISoap11Port" binding="HelloWorldAPISoap11Binding">
        <soap11:address location="http://localhost:8180/pc/ws/example/HelloWorldAPI"/>
        <gw:address location="${pc}/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 PolicyCenter web service from Java

For web services, there are no automatically generated libraries to generate 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. There are several ways to create Java classes from the WSDL:

- The Java API for XML Web Services (JAX-WS) includes a tool called `wsimport` that generates Java-compatible stubs from a WSDL file.
- The CXF open source tool.
- The Axis2 open source tool.

Calling web services using Java and wsimport

The Java API for XML Web Services (JAX-WS) includes a tool called `wsimport` that generates Java-compatible stubs from a WSDL file. This documentation uses the `wsimport` tool and its output to demonstrate creating client connections to the PolicyCenter web services.

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

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 `SUBDIRECTORY_NAME`.

5. Type the following command:

```
wsimport WSDL_LOCATION_URL -s SUBDIRECTORY_NAME
```

If the `SUBDIRECTORY_NAME` 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:PORTNUMBER/pc/ws/example/HelloWorldAPI?WSDL -s src
```

6. The tool generates Java source files and compiled class files. Depending on what you are doing, you probably need only the class files or the source files, but not both.

a. The `.java` files are in the `SUBDIRECTORY_NAME` subdirectory. To use them, add this directory to your Java project's class path, or copy the files to your Java project's `src` folder.

The location of the files represents the hierarchical structure of the web service namespace in reverse order, followed by the fully qualified name.

The namespace is a URL that each published web service specifies in its declaration. It represents a namespace for all the objects in the WSDL. A typical namespace would specify your company domain name and perhaps other meaningful disambiguating or grouping information about the purpose of the service. For example, `http://mycompany.com`.

You can specify the namespace for each 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 is `http://example.com`.

The path to the Java source file has the following structure.

```
CURRENT_DIRECTORY/SUBDIRECTORY_NAME/REVERSED_NAMESPACE/FULLY_QUALIFIED_NAME.java
```

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

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

- b. To use the generated Java `.class` files, add the generated directory to your Java project's class path, or copy the files to your Java project's `src` folder.

The compiled `.class` files are placed in a hierarchy by package name with the same basic naming convention as the `.java` files but with no `SUBDIRECTORY_NAME`.

```
CURRENT_DIRECTORY/SUBDIRECTORY_NAME/REVERSED_NAMESPACE/FULLY_QUALIFIED_NAME
```

The `wsimport` tool generates the `.class` files at the following location:

```
CURRENT_DIRECTORY/com/example/com/mycompany/HelloWorld.class
```

7. The next step depends on whether you want just the `.class` files to compile against, or whether you want to use the generated Java files.
 - To use the `.java` files, copy the `SUBDIRECTORY_NAME` 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.
8. 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.

```
HelloWorldAPIPortType 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 48. 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.HelloWorldAPIPortType;

public class WsiTestNoAuth {

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

        // Get a reference to the SOAP 1.1 port for this web service
        HelloWorldAPIPortType 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 PolicyCenter web service.

```
import com.example.example.helloworldapi.HelloWorldAPI;
import com.example.example.helloworldapi.HelloWorldAPIPortType;
import com.sun.xml.internal.ws.api.message.Headers;
import 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
HelloWorldAPIPortType 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, PolicyCenter 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 WsTest01 {

    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 PolicyCenter 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 PolicyCenter installation directory, run the following command.

```
gwb genWsiLocal
```

PolicyCenter 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.

```
PolicyCenter/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. PolicyCenter includes with WSDL for some local services in the default configuration. These WSDL files are in Studio modules other than `configuration`. If PolicyCenter 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, PolicyCenter 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 PolicyCenter users whose express purpose is to provide authentication. You can create fictional users by using the PolicyCenter 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 PolicyCenter 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 PolicyCenter 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 PolicyCenter 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 189

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 PolicyCenter database. If authentication could not be attempted, such as if network problems existed, the method returns `null`. Other causes of authentication failure throw a `WsiAuthenticationException`.

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, PolicyCenter 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 70

Transforming a generated schema

PolicyCenter 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.xmleschema.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 PolicyCenter implementations, a web service is implemented by a Gosu class with each method implementing one web service operation. PolicyCenter 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.

```
PolicyCenter/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.

See also

- “Example of an invocation handler for preexisting WSDL” on page 63

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 PolicyCenter 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 PolicyCenter 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 PolicyCenter 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 PolicyCenter 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 PolicyCenter 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 PolicyCenter 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” on page 60.

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 72

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 91

Checking for duplicate external transaction IDs

To detect duplicate operations from external systems that change data, add the `@WsiCheckDuplicateExternalTransaction` 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, PolicyCenter checks for the SOAP header `<transaction_id>` in namespace `http://guidewire.com/ws/soapheaders`. If the SOAP header `<transaction_id>` is missing, PolicyCenter throws an exception.

PolicyCenter has some web services that include a transaction ID as a method parameter explicitly. These methods do not use the `@WsiCheckDuplicateExternalTransaction` annotation or its built-in functionality.

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 89.

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, PolicyCenter 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 PolicyCenter server runs in only one environment. In the case of a server cluster, PolicyCenter requires that all servers in the cluster 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.ws1`, then the service has the fully-qualified type `examples.gosu.ws1.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.p1.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.IWsWebServiceConfigurationProvider`. 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.IWsWeiwserviceConfigurationProvider

class ABConfigurationProvider implements IWsWeiwserviceConfigurationProvider {

    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

PolicyCenter 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"/>-->  
</suite-config>
```

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"/>-->  
</suite-config>
```

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, PolicyCenter does the following.

1. Checks to see whether there is a WSDL port of element type **<gwwsdl:address>**. If such a port is found, PolicyCenter ignores any other ports on the WSDL for this service.
2. If so, it looks for shortcuts with the syntax **\${productNameShortcut}** . For example: **\${pc}** .
3. If that product name shortcut is in the **suite-config.xml** file, PolicyCenter 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 PolicyCenter the abbreviation is **pc**.

```
<wsdl:port name="TestServiceSoap11Port" binding="TestServiceSoap11Binding">
<soap11:address location="http://172.24.11.41:8480/pc/ws/gw/test/TestService/soap11" />
<gwwsdl:address location="${pc}/ws/gw/test/TestService/soap11" />
</wsdl:port>
```

Accessing the suite configuration file by using the API

PolicyCenter 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 `getProductInfo` 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.getProductInfo("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, PolicyCenter 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.

PolicyCenter 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.wsidupdatefaults.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.wsdl.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 93

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.wsdl.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, PolicyCenter 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.testharness.TestBase
uses gw.testharness.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.

General web services

This topic describes general-purpose web services, such as mapping typecodes general system tools.

Importing administrative data

PolicyCenter 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.

```
PolicyCenter/build/xsd/pc_import.xsd  
PolicyCenter/build/xsd/pc_entities.xsd  
PolicyCenter/build/xsd/pc_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. PolicyCenter 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 PolicyCenter, not additional Guidewire applications that you integrated with PolicyCenter. In particular, batch process methods called on a PolicyCenter server apply only to PolicyCenter 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 *System 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.pc.pc1000.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 PolicyCenter:

```
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.

```
waitForActiveWorkItems(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, ActivityEsc, 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.

See also

- “maintenance_tools command options” in the *System 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 *System Administration Guide*

Mapping typecodes and external system codes

In the simplest case, the typecodes used by PolicyCenter 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 `PolicyCenter/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 PolicyCenter typelist typecode of `LossType.PR` to its equivalent code in two external systems, `ExtSys123` and `ExtSysABC`.

```
<?xml version="1.0"?>
<typecodemapping>
  <nspacelist>
    <namespace name="ExtSys123"/>
    <namespace name="ExtSysABC"/>
  </nspacelist>

  <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 PolicyCenter typecode to the relevant external system code. The mapped PolicyCenter 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 PolicyCenter 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. PolicyCenter 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 PolicyCenter 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 PolicyCenter 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 PolicyCenter. The methods enable the external system to translate one of its own codes to a mapped PolicyCenter 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 PolicyCenter 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 PolicyCenter. The term "Alias" in the method names refers to an external system's code, and "InternalCode" refers to a PolicyCenter typelist and typecode combination. The methods enable the external system to translate a PolicyCenter 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 PolicyCenter 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 PolicyCenter 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 PolicyCenter 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 PolicyCenter 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 PolicyCenter 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 PolicyCenter typelist and typecode combination. The methods enable the configuration code to translate a PolicyCenter 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 PolicyCenter 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 PolicyCenter is importing data from an external system. The methods enable PolicyCenter 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 PolicyCenter 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`.

The Profiler API web service

The `ProfilerAPI` web service enables or disables the PolicyCenter 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.

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 PolicyCenter server using the following syntax.

```
http://server:port/pc/ping
```

An example command line is shown below.

```
http://PolicyCenter.mycompany.com:8180/pc/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 0. A null character result might not be returnable for some combinations of HTTP servers and clients.	GW_STARTING

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`. PolicyCenter 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 PolicyCenter 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. PolicyCenter 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>clearErrorHandler</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.

Action	TableImportAPI method and Description
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 PolicyCenter. 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.
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 PolicyCenter 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	invokeTrigger 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	isTriggerAvailable 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	resumeWorkflow 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	resumeAllWorkflows 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	suspend 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	complete 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 PolicyCenter 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. PolicyCenter 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.
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.

Account and policy web services

PolicyCenter provides web services that enable you to accept changes to accounts and policies from external systems.

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`
- `XSDTTime`

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`.

Account web services

Use the `AccountAPI` web service to add documents to accounts, add notes to accounts, and find account public IDs.

Add documents to an account

Use the `addDocumentToAccount` method to add a document to an account. The method is relatively straightforward, taking an account's public ID and a `Document` object and adds the document to the account. PolicyCenter returns the public ID of the newly added document.

Add notes to an account

Use the `addNoteToAccount` method to add a note to an account. The method is relatively straightforward, taking an account's public ID and a `Note` object and adds the document to the account. PolicyCenter returns the public ID of the newly added note.

Find accounts

There are two ways to find accounts, either by account number or by complex criteria.

Use the `findAccountPublicIdByAccountNumber` method to find an account's public ID given its account number. It returns the public ID of the account, or `null` if there is no such account.

```
accountPublicID = accountAPI.findAccountPublicIdByAccountNumber("ABC-00001");
```

In contrast, the `findAccounts` method is more robust and allows you to search for multiple accounts that fit certain criteria. For example, you could find accounts that match a certain account status or from a certain city. To use it, create a new `AccountSearchCriteria` object, fill in properties that match what you want to search for, and pass it to `findAccounts`.

```
AccountSearchCriteria acs = new AccountSearchCriteria();
acs.setCity("San Francisco");
accounts = accountAPI.findAccounts(acs);

// get first result in the array...
accountPublicID = accounts[0];
```

Assign activities

You can remotely add and assign an activity for an account by using the `addAndAutoAssignActivityForAccount` method. PolicyCenter generates an activity based on the activity type and the activity pattern code and then auto-assigns the activity.

It takes a `String` account public ID, an `ActivityType` object, a `String` activity pattern code, a `String` activity pattern code, and an array of activity properties of type `ActivityField`. The method has no return value.

Add contacts to an account

You can remotely add a contact to an account by using the `addContactToAccount` method. It takes a `String` account public ID and an account contact of type `AccountContact`.

The method adds the given account contact to the account, and returns the public ID of the new `AccountContact`.

Add location to an account

You can remotely add a location to an account by using the `addLocationToAccount` method. It takes a `String` account public ID, an `AccountLocation` object, and returns the `String` public ID of the new `AccountLocation` entity.

Retrieve account numbers

You can remotely get the location to an account by using the `getAccountNumber` method. It accepts a `String` account public ID and returns the associated `String` account number.

Check for active policies

You can remotely check for active policies for the account by using the `hasActivePolicies` method. A policy is considered active if it has any `PolicyPeriod` entities that are issued and are effective at the current date and time.

The method accepts a `String` account public ID. The method returns `true` if the account has active policies, otherwise, `false`.

Insert accounts

You can remotely insert an account by using the `insertAccount` method. The method accepts an `ExternalAccount` entity, which is a parallel to the `Account` entity. The `ExternalAccount` entity contains a subset of the properties on the internal non-SOAP `Account` entity due to the necessity to simplify the entity for serialization across the SOAP protocol.

After importing the external account into PolicyCenter as an `Account` entity, PolicyCenter returns the resulting `String` public ID of the new `Account` entity.

Merge accounts

You can remotely merge two accounts by using the `mergeAccounts` method. This task is available from this web service but is not provided in the PolicyCenter reference implementation in the user interface. You must decide which account is the merged account and which account to discard after the merge is complete.

The method takes two `String` account public IDs. The first public ID represents the source account to be copied then discarded afterward. The second public ID represents the destination account to have the source destination account merged with it. The method returns nothing.

See also

- To customize the operations that are performed after merging accounts, see “Account plugin” on page 153.

Transfer policies

You can remotely transfer policies from one account to another by using the `transferPolicies` method. Provide an array of `String` policy public IDs, the public ID of the source account, and the public ID of the destination account.

See also

- To customize the operations that are performed after transferring accounts, see “Account plugin” on page 153.

Update account contacts

You can remotely update the contact information for account contacts by using the `updateAccountContact` method. You provide a `String` account public ID, a `String` contact display name, and account contact information encapsulated in an `AccountContactInfo` entity. Notice that this argument is the `AccountContactInfo` entity and not the `AccountContact` entity. `AccountContactInfo` simply encapsulates an `Address` entity in its

`PrimaryAddress` property. Populate an `Address` entity with the new information and set the `PrimaryAddress` property on the `AccountContactInfo` entity.

PolicyCenter looks at the specified account and finds the contact with the provided display name. PolicyCenter then updates the contact with the information in the `AccountContactInfo`.

Job web services

Use the `JobAPI` web service to add activities to jobs or to withdraw preempted jobs.

The primary methods of this web service add new activities to jobs. They differ in how groups and users are assigned new activities.

- `addActivityFromPatternAndAssignToUser`
- `addActivityFromPatternAndAssignToQueue`
- `addActivityFromPatternAndAutoAssign`

Specify the activities that you add using activity patterns. Users define activity patterns in the administrative data of your PolicyCenter instance. An activity pattern is a template that sets default values for new activities.

Common parameters in the job APIs

The common parameters for the primary methods are listed below.

- Public ID of the job. To get the public ID, use the helper method `findJobPublicIdByJobNumber`.
- Public ID of the activity pattern. To get the public ID, use the method `ActivityPattern.finder.getActivityPatternByCode("pattern-code")`.
- Activity fields, as a `gw.webservice.pc.pc1000.gxmodel.activitymodel.types.complex.Activity` object. Use an instance of the XML type `Activity` to initialize activity fields that activity patterns do not cover.

Note the following characteristics about the parameters to the Job APIs.

- Activity patterns must not be designated “automated only.”
- Activity patterns must be available for the type of jobs to which you try to add the activities. Otherwise, Guidewire throws the exception `EntityStateException`.
- Users of the web services user must have the permissions `VIEW_JOB` and `CREATE_ACTIVITY`. Otherwise, Guidewire throws the exception `PermissionException`.
- Parameters, such job IDs, activity pattern IDs, user IDs, queue IDs, and group IDs, must refer to existing entity instances. Otherwise, Guidewire throws the exception `DataConversionException`.

Helper methods of the job APIs

The `JobAPI` web service provides the following helper methods.

- `findJobPublicIdByJobNumber` – Given a job number, returns the public ID of the job. Use the public ID as a parameter to other methods of the Job APIs.
- `withdrawJob` – Withdraws preempted jobs.

Policy cancellation and reinstatement web services

Use the `CancellationAPI` and the `ReinstatementAPI` web services to start policy cancellation and policy reinstatement jobs. These web services are WS-I compliant.

PolicyCenter starts some types of cancellations.

- Cancellation as part of a rewrite request
- Cancellation at the insured’s request
- Cancellation because the insured provided false information in their application or violated the contract

- Cancellation because the insured chooses not to take a renewal and the renewal might already be bound

There are other types of cancellations that the billing system or other integration systems initiate. For example, cancellation for non-payment. If the billing system detects overdue invoices, the billing system starts a delinquency process and tells PolicyCenter to cancel the policy.

There are two ways the billing system can tell PolicyCenter to cancel a policy.

- Cancel the policy as of a certain date – PolicyCenter sets the policy to pending cancellation within PolicyCenter and cancels automatically on the given date unless the application receives a rescind command before that date.
- Cancel the policy as soon as possible – This is similar to the previous approach described but PolicyCenter uses internal notification lead time logic to determine the actual cancellation date.

PolicyCenter supports both approaches based on the cancellation date you pass to the API.

See also

- For more information about circumstances in which PolicyCenter might trigger cancellation or reinstatement, refer to “Policy period management” on page 535.

Begin a cancellation

An external system can begin a cancellation in PolicyCenter with the `beginCancellation` method on `CancellationAPI`.

```
beginCancellation(String policyNumber, Calendar cancellationDate, boolean recalculateEffDate,
    CancellationSource cancellationSource, ReasonCode reasonCode, CalculationMethod refundCalcMethod,
    String description, String transactionId)
```

To begin cancellation, you must specify a reason in a `ReasonCode` enumeration to indicate fraud, eligibility issues, and so on. Instead of passing the reason code directly, a `CancellationSetup` entity encapsulates this enumeration.

The method returns the public ID of the cancellation job.

The parameters are described below.

- Policy number, as a `String`
- Cancellation date, as a `Calendar` object
- Recalculate effective date (`recalculateEffDate`), as a boolean
- Cancellation source typecode, `carrier` (insurer initiated cancel) or `insured` (the insured initiated cancel)
- Reason code typecode, as a `ReasonCode` value
- Refund calculation method, as a `CalculationMethod`
- Description (notes) associated with this cancellation, as a `String`
- Transaction ID to associate with this change

If you set `recalculateEffDate` to `false`, starts a Cancellation job effective on the cancellation date, or the given policy in PolicyCenter (this date will also be the effective date of the new cancellation policy period). In this case, the caller must check for any legal requirements for the cancellation date. If `recalculateEffDate` is set to true, PolicyCenter calculates the earliest date for the cancellation to meet all legal requirements. PolicyCenter uses that date if it is after the cancellation date. In this case, the caller can set the `cancellationDate` to today to get the policy period cancelled as soon as possible.

The following Java example cancels a policy.

```
String jobNumber = cancellationAPI.beginCancellation( "ABC:123",
    cancellationDate.toCalendar(), true, CancellationSource.TC_CARRIER,
    ReasonCode.TC_NONPAYMENT, CalculationMethod.TC_PRORATA, "Some Description", "External1235")
```

Rescind a cancellation

Rescind a cancellation with the `rescindCancellation` method.

The method has two versions.

- One version rescinds a policy cancellation with a reason as a `ReasonCode` typecode and a cancellation source as a `CancellationSource` typecode. The reason typecode indicates a payment-related reinstatement with the value `payment` or another reason with the value `other`. The cancellation source typecode indicates that the insurer initiated the cancellation with the value `carrier` or the insured initiated the cancellation with the value `insured`.

```
void rescindCancellation(java.lang.String policyNumber,
    java.util.Calendar effectiveDate, CancellationSource cancellationSource, ReasonCode reasonCode)
```

For example, the following sample code rescinds a cancellation with a reason code.

```
cancellationAPI.rescindCancellation("ABC:123", new Date(),
    CancellationSource.TC_CARRIER, ReasonCode.TC_NONPAYMENT)
```

- The other version rescinds an in-progress policy cancellation with the job number of a cancellation. For example, the following sample code rescinds a cancellation with a job number only.

```
void rescindCancellation("ABC:123")
```

Find a cancellation

In some cases, an external system wants to rescind a cancellation but does not have the job number for the cancellation. In these cases, use the `findCancellations` method on the `CancellationAPI` web service to get the job number. The method returns a list of job numbers for matching cancellations. You can use these results to decide which cancellation you want to rescind. Next, call the separate `rescindCancellation` method and pass it the job number of the cancellation job to rescind.

The `findCancellations` method takes the following arguments in the following order.

- Policy number, as a `String`
- Cancellation date, as a `java.util.Calendar` object. PolicyCenter finds cancellations effective on that date or after that date.
- Cancellation source, as a `CancellationSource` typecode
 - `carrier` – Cancellation initiated by insurer
 - `insured` - Cancellation initiated by insured
- Reason code, as a `ReasonCode` typecode
 - `nonpayment` – Payment not received
 - `fraud` – Fraud
 - `flatrewrite` – Policy rewritten or replaced (flat cancel)
 - `midtermrewrite` – Policy rewritten (mid-term)
- Calculation method, as a `CalculationMethod` typecode
 - `flat` – Flat
 - `prorata` – Pro rata
 - `shortrate` – Short rate

The following example code demonstrates how to call this method from Java.

```
Calendar cancellationDate = Calendar.getInstance();
String jobNumber;
String myPolicyNumber = "abc:123";
var foundJobNumbers = cancellationAPI.findCancellations(myPolicyNumber, cancellationDate,
    CancellationSource.TC_carrier, ReasonCode.TC_nonpayment, CalculationMethod.TC_prorata);
```

Reinstate a policy

Use the `ReinstatementAPI` web service to start a reinstatement job for a policy that is canceled and that you want to put it back in-force. The reinstatement effective date is always the cancellation date. To create a gap in coverage, a user must start a policy rewrite job manually through the PolicyCenter interface.

The `ReinstateAPI` web service has a single method called `beginReinstatement`. The method accepts the following arguments.

- Policy number as a `String`
- Reinstatement code as a `ReinstateCode` typecode

The method returns a job number for the new reinstatement job.

The following example code demonstrates how to call this from Java.

```
reinstatementAPI.beginReinstatement( "ABC:1234", ReinstateCode.TC_payment );
```

Submission web services

Use the `SubmissionAPI` web service to start draft and quote submissions for policy renewals from external systems.

Passing in external policy period data for submissions

To start submission jobs in PolicyCenter, the `SubmissionAPI` web service accepts data that populates PolicyCenter policy periods, which submission jobs use. The entire set of external data comes in the `policyPeriodData` parameter, defined as a `String`. You can use any text-based format in your implementation that you can parse, for example XML format.

To parse the `policyPeriodData` parameter, the methods on the `SubmissionAPI` web service parse the `String` data as XML by using the Guidewire XML model (GX model) definition. The GX Model is available at the location `gw.webservice.pc.pc1000.gxmodel.policyperiodmodel`. Then the code calls the `populatePolicyPeriod` enhancement method on the XML object. The enhancement method is implemented in `gw.webservice.pc.pc1000.gxmodel.PolicyPeriodModelEnhancement`.

If you want to add line-specific populating or validation code, modify the `populatePolicyPeriod` enhancement method. If you need to use a different XSD or GX model, modify the `SubmissionAPI` web service implementation class directly.

Start a submission

The `startDraftSubmission` method creates a submission and leaves it in a draft state.

```
function startDraftSubmission(accountNumber : String, productCode : String, producerCodeId : String,  
    policyPeriodData : String, parseOptions : String) : String
```

The method accepts the following arguments.

- Account number as a `String`
- Product code as a `String`
- Producer code public ID as a `String`
- XML representation of policy period data from the external system as a `String`
- Set of parse options as a `String` that controls how your implementation parses the policy period data

The method returns a `String` that contains the job number of the submission.

Start a submission and generate a quote

The `quoteSubmission` method creates a submission and attempts to generate a quote.

```
function quoteSubmission(accountNumber : String, productCode : String, producerCodeId : String,  
    policyPeriodData : String, parseOptions : String) : String
```

The method accepts the following arguments.

- Account number as a `String`
- Product code as a `String`
- Producer code public ID as a `String`
- XML representation of policy period data from the external system as a `String`
- Set of parse options as a `String` that controls how your implementation parses the policy period data

The method returns a `String` that contains the job number of the submission.

Producer web services

The `ProducerAPI` web service manages producer codes, agencies, and branches. Agencies are a type of organization. Branches are a type of group. Branch objects are distinguished from regular group objects by having a non-null value for the `BranchCode` attribute.

Refer to the WS-I implementation class in Studio for details of the methods on this web service.

```
gw.webservice.pc.pcVERSION.community.ProducerAPI
```

Remember the following characteristics about producer hierarchies.

- Note that each `Organization` in turn may have an associated tree of `Group` entities. For a `Group`, ancestors means the group's ancestors in its `Group` tree. Groups and producer codes can be arranged hierarchically, but organizations, including agencies, are not arranged hierarchically. For this web service, uses of the `PolicyCenter Organization` entity use an instance of `OrganizationDTO`, which is a `Gosu` class in package `gw.webservice.pc.pcVERSION.community.datamodel`.
- Every `ProducerCode` is associated with a single `Branch`, and there is a many-to-many correspondence between producer codes and agencies and their associated groups.
- The `Agency-ProducerCode` constraint limits the associations between agencies and producer codes.

Product Model web services

You can use the `PolicyCenter` product model web service `ProductModelAPI` to do several things.

- Update the `PolicyCenter` product model on a running development server from the local file system XML representation of the product model. The product model lists all the insurance products that an insurance company sells. Typically, you would manage the product model using the `Product Designer` application.
- Update system tables on a running development server based on local file system representation of the system tables.
- Query the server for product model information. Generally speaking, it is best to identify product model objects using the property `CodeIdentifier`, not `PublicID`. `ProductModelAPI` web service primarily identifies product model objects using the property `PublicID`. From an external system, you can get the `PublicId` for a product model object by calling the `ProductModelAPI` web service method `getPublicIdForCodeIdentifier`. If you develop your own web services that manipulate product model types, Guidewire recommends using the property `CodeIdentifier` for arguments and return values.

Synchronize Product Model in database with file system XML

The most important method in the `ProductModelAPI` web service is the `synchronizeProductModel` method. Use the `synchronizeProductModel` method to update the `PolicyCenter` product model in memory from the local file system XML representation of the product model.

The `Product Designer` application uses this API to reload the `PolicyCenter` server in-memory product model after modifying the local XML files.

Suppose you plan to introduce a new insurance product such as a new type of auto insurance coverage. You may want to change the PolicyCenter product model in multiple ways. However, never push product model changes directly to the production server.

Instead, test changes on different instances of the development server. Ensure your product model changes are correct before deploying to the production server.

To maintain business data integrity, PolicyCenter forbids some types of changes to product model entity properties on production servers. Some product model changes are impossible to undo after you deploy the changes to a production server. For example, a coverage term with the value 100 cannot change to the value 200 on a production server. That would fundamentally change the meaning of any policies that used that coverage term. This is a difference between working with a development server which has no customer data and a production server where customer data is bound by legal and customer service implications.

The development server must have an identical PolicyCenter configuration as the production server, except for the following configuration settings.

- On the production server, set the server environment property `env` to `prod` (production).
- On the development server, set the server environment property `env` to a value other than `prod`.

Fully test product model changes on a development server. PolicyCenter protects the business and legal integrity of customer data in a special way for production servers. Deploying unacceptable property changes to existing product model entities prevents PolicyCenter from starting in production environment.

If you call the `ProductModelAPI` web service on a server whose system environment variable `env` is set to the value `prod`, the web service throws an exception.

Use this API in the following cases.

- If you need to revert the product model to the product model XML files stored on the server instead of the data persisted in the database.
- If you change a development server's local file resources in some other way during development due to source control changes or hand-editing of files.

Use this API if you change a development server's local file resources in some other way during development due to source control changes or hand-editing of files.

Do not use this API if the file did not change since server startup.

Synchronize system tables

Another `ProductModelAPI` web service method is `synchronizeSystemTables`. Use the `synchronizeSystemTables` method to modify the PolicyCenter database's system tables from the local file system XML representation of system tables. This method takes no arguments, and returns nothing.

The Product Designer application uses this API to reload the database system tables into PolicyCenter after modifying the local XML files.

If you call this API on a server whose system environment variable `env` is set to the value `prod`, the web service throws an exception.

Use this API if you change a development server's local file resources in some other way during development due to source control changes or hand-editing of files.

Do not use this API if the files did not change since server startup.

Query the database for Product Model information

On a running PolicyCenter server, you can query the database for product model information.

There are multiple types of queries you can perform.

- To return the list of available questions for a specified policy period, call the `ProductModelAPI` web service method `getAvailableQuestions`. The return type is a list of question sets, as the type `List<QuestionSet>`.
- To return the list of available clauses (such as coverages, conditions, and exclusions) for a specified policy period, call the `ProductModelAPI` web service method `getAvailableClausePatterns`. The return type is a list of clause patterns, as the type `List<ClausePattern>`.

The arguments are the same for both methods.

- *LookupRoot* – the information about the entity to search for availability, as a `LookupRootImpl` object. The `LookupRootImpl` object encapsulates the search criteria and contains the following properties:
 - `LookupTypeName` – the `String` lookup type name
 - `PolicyLinePatternCode` – the `String` policy line pattern code
 - `CovTermPatternCode` – the `String` coverage term pattern code
 - `ProductCode` – the `String` product code
 - `JobType` – the job type as a `Job` typekey
 - `Jurisdiction` – a jurisdiction as a `Jurisdiction` object
 - `PolicyType` – a policy type as a `BAPolicyType` object
 - `UWCompanyCode` – an underwriter company code as a `UWCompanyCode` object
 - `IndustryCode` – a `String` industry code
 - `VehicleType` – a vehicle type as a `VehicleType` object
- *offeringCode* – a `String` offering code
- *LookupDate* – the date to look up as a standard `Date` object

Exceptions that the methods may throw are listed below.

- `SOAPException` – If communication fails
- `RequiredFieldException` – If any required field is null
- `BadIdentifierException` – If the API cannot find an instance with specified ID

Policy web services

To control policy period referral reasons and activities from an external system, use the `PolicyAPI` web service.

Add a referral reason

The `PolicyAPI` method `addReferralReasonToPolicyPeriod` adds a period referral reason to the policy period. For example, if a claim management system notices a lot of claims on a policy, it could notify PolicyCenter of the issue. PolicyCenter could associate the issue with the appropriate policy period. The renewal workflow could set risk-related properties on the policy period before authorizing renewal of that policy.

```
function addReferralReason(policyId : String, issueTypeCode : String, issueKey : String,
                           shortDescription : String, longDescription : String, value : String) : String
```

The method accepts the following arguments.

- Policy ID (`String`)
- Issue type code (`String`)
- Issue key (`String`)
- Short description (`String`)
- Long description (`String`)
- Referral reason, which must be valid for the comparator of the `UWIssueType`.

The method returns the public ID of the new or existing referral reason. PolicyCenter sets the status of the new referral reason to Open.

Close a referral reason

The `closeReferralReason` method closes a referral reason.

```
function closeReferralReason(policyId : String, issueTypeCode : String, issueKey : String)
```

The method accepts the following arguments.

- Policy period public ID
- Code that matches the `UWIssueType.Code` of the referral reason to close
- Issue key identifier for the referral reason

The method has no return value.

Add activity from pattern

There are several methods for adding activities from patterns and assigning them.

Add activity and auto-assign

To add an activity using an activity pattern from an external system and auto-assign the activity, call the `IPolicyAPI` web service method `addActivityFromPatternAndAutoAssign`.

```
function addActivityFromPatternAndAutoAssign(policyId : String, activityType : ActivityType,  
    activityPatternCode : String, activityFields : Activity) : String {
```

The `activityfields` argument contains an `Activity` object, which is not the Guidewire entity. Instead, it is a XML type defined using the Guidewire XML (GX) modeler.

Set the important properties for a new activity, such as the `Subject` and the `Description` properties. PolicyCenter copies over non-null fields from the `activityfields` argument to the new activity.

The `addActivityFromPatternAndAutoAssign` method performs the following operations.

1. Try to generate an activity from the specified activity pattern.
2. Initialize the activity with the following fields from the activity pattern: `Pattern`, `Type`, `Subject`, `Description`, `Mandatory`, `Priority`, `Recurring`, and `Command`.
3. Calculate the target date using the following properties from the pattern: `TargetStartPoint`, `TargetDays`, `TargetHours`, and `TargetIncludeDays`.
4. Calculate the escalation date using the following properties from the pattern `EscalationStartPt`, `EscalationDays`, `EscalationHours`, and `EscalationInclDays`. If the activity does not use those properties, PolicyCenter does not set the target and/or escalation date. If the target date after the escalation date, then PolicyCenter sets the target date to the escalation date.
5. PolicyCenter sets the activity policy ID to the specified policy ID. The previous user ID for the activity (the `previousUserId` property) is set to the current user.
6. PolicyCenter assigns the newly created activity to a group and/or user using the Assignment Engine.
This step is the auto-assignment step.
7. PolicyCenter saves the activity in the database.
8. The method returns the ID of the newly created activity.

Add activity and assign to user

If you want to assign the new activity to a specific group and user, call the `addActivityFromPatternAndAssignToUser` method.

```
function addActivityFromPatternAndAssignToUser(policyId : String, userId : String, groupId : String,  
    activityType : ActivityType, activityPatternCode : String, activityFields : Activity) : String {
```

The method performs the same operations as the related `addActivityFromPatternAndAutoAssign` method, except that it assigns the activity to the specified group and user.

Add activity and assign to queue

If you want to assign the new activity to a specific group and user, call the `addActivityFromPatternAndAssignToQueue` method.

```
function addActivityFromPatternAndAssignToQueue(policyId : String, queueId : String,  
activityType : ActivityType, activityPatternCode : String, activityFields : Activity) : String {
```

The method performs the same operations as the related `addActivityFromPatternAndAutoAssign` method, except that it assigns the activity to the specified queue.

Policy period web services

To add notes to policies and policy periods and to add documents to policy periods, use the `PolicyPeriodAPI` web service.

The methods of the `PolicyPeriodAPI` web service all accept the public IDs of policies and policy periods as `String` parameters. In addition, they accept instances of GX models for notes or documents as `String` parameters. A GX model contains an entity's property data in XML format.

Add notes to policies and policy periods

Use the `addNoteToPolicy` method to add a note to a policy. It takes the public ID of a policy, as a `String`, and a GX model of a `Note`.

Use the `addNoteToPolicyPeriod` method to add a note to a policy period. It takes the public ID of a policy period, as a `String`, and a GX model of a `Note`.

Add documents to policy periods

To add a document to a policy period, call the `PolicyPeriodAPI` web service method `addDocumentToPolicyPeriod`. The method accepts the following arguments.

- Public ID of a policy period
- GX model of a Document.

The method returns the public ID of the new `Document` entity instance.

Policy earned premium web services

To calculate earned premiums from an external system, use the web service `PolicyEarnedPremiumAPI`. It has a single method called `calcEarnedPremiumByPolicyNumber`, which calculates the earned premium for a given policy number. The method accepts the following arguments.

- A `String` Policy number
- A `Date` object on which the policy to find is in effect. If the policy has multiple terms, the date determines which policy term to use. The method selects the most recent version of the policy for the given term.
- The date as of which to calculate earned premium amounts
- A Boolean value (`includeEBUR`) that specifies whether to include data that is earned but not reported (EBUR).

The method returns a list of `PolicyEarnedPremiumInfo` objects, one for each line of business on the policy. Each `PolicyEarnedPremiumInfo` contains two properties.

- `LOB` – the line of business code as a `String` value
- `EarnedPremium` – the earned premium as a `BigDecimal` value

For EBUR data, it is standard practice for premium reporting policies to use actual reported premium rather than a pro rata estimate of estimated premium as the basis for earning. However, suppose PolicyCenter received monthly premium reports through the end of May (received June 15) and now wants to do June month-end earning accruals as of July 5. There is no report for premiums earned in June, because it is not expected until July 15). If you include EBUR data, PolicyCenter estimates how much of the policy premiums are earned during this period of elapsed coverage but with no actual premium report. After the last report is received or a final audit is complete, no portion of the period remains unreported.

The `includeEBUR` parameter interacts in a special way with the status of the policy as a reporting policy.

- If `includeEBUR` is `true` and the policy is not a reporting policy, the `calcEarnedPremiumByPolicyNumber` method returns an error.
- If the policy is a reporting policy but final audit is complete, PolicyCenter treats the `includeEBUR` parameter as if it were `false` even if you passed `true`.

Import policy web services

To import policies from XML data from an external system, use the `PolicyChangeAPI` web service.

There is one method called `quoteSubmission`, which you can call to quote a submission in PolicyCenter.

The `quoteSubmission` method accepts the following arguments.

- An account number
- The product code, such as `PersonalAuto` or `WorkersComp`
- A producer code public ID
- A policy period data object, which is XML data
- Parse options to pass to XML parser

Within PolicyCenter, the format of the XML must conform to the structure defined by the GX model in the following file.

```
gw.webservice.pc.pcVERSION.gxmodel.policyperiodmodel
```

The method returns an object of type `QuoteResponse`, which is a Gosu class that contains the following properties.

- `JobNumber` – Job number of the new job
- `Errors` – Any errors as an array of `String` objects

Policy change web services

To start policy change jobs from an external system, use the `PolicyChangeAPI` web service.

Policy change jobs modify in-force policies. Policy changes range from simple location changes to complex coverage changes. Policy change jobs often require new quotes and bindings of policy periods.

The `PolicyChangeAPI` web service has two methods. Both take the following parameters:

- Policy number as a `String`
- Effective date as a `Date`

Both methods return the `JobNumber` property of the new job. The methods differ in whether the policy change is quoted and bound automatically by the web service or manually by a user.

See also

- *Application Guide*

Starting an automatic policy change job

Use the `startAutomaticPolicyChange` method to start a policy change job that attempts to quote the change and bind the policy automatically. If errors occur, users complete the policy changes manually through the PolicyCenter interface.

You can modify the default behavior by changing the implementation of `PolicyChangeProcess.startAutomatic`.

An actual workflow can be started with this method to control the job's progress.

Starting a manual policy change job

Use the `startManualPolicyChange` method to start a policy change job that waits in draft mode for a user to quote and finish it manually. The method does not attempt to quote and bind the change automatically.

Specifying what policy data to change

In the default configuration, the methods of the `PolicyChangeAPI` web service do not change any policy data. You must add your own methods to change policy data.

PolicyCenter updates policy contacts from account contacts automatically after a job starts. You can use this API to trigger a policy change job. The API pulls in the latest version of each `PolicyContact` object from its associated `AccountContact` object.

Policy renewal web services

The `PolicyRenewalAPI` handles policy renewal requests received from external systems. The service is located in the package `gw.webservice.pc.VERSION.job`.

Start renewals on existing policies

The `startRenewals` method of the `PolicyRenewalAPI` web service starts renewals on one or more existing PolicyCenter policies.

```
startRenewals(policyNumbers : String[])
```

The `policyNumbers` argument is an array of policy numbers that reference existing policies in PolicyCenter. If a referenced policy does not exist in PolicyCenter, a `BadIdentifierException` is thrown and any subsequent policies in the array are not processed.

The method has no return value.

The method starts a renewal job for each referenced policy. The entries in `policyNumbers` are processed in sequence. When a successful renewal is created, it is committed to the database. No gap in coverage exists between the original policy period and its renewal.

Import a policy for renewal

The `startConversionRenewal` method of the `PolicyRenewalAPI` web service starts a renewal for a single policy. The policy does not yet exist in PolicyCenter, but is imported from an external system as part of the method's renewal process.

```
startConversionRenewal(accountNumber : String,
                      productCode : String,
                      producerCodeId : String,
                      policyNumber : String,
                      policyPeriodData : String,
                      changesToApply : String,
                      parseOptions : String,
                      basedOnEffectiveDate : String,
                      basedOnExpirationDate : String) : String
```

The method accepts the following arguments.

- `accountNumber` - Reference to an existing PolicyCenter account. If the account does not exist in PolicyCenter, a `BadIdentifierException` is thrown.
- `productCode` - Reference to an existing PolicyCenter product code, such as `PersonalAuto` or `WorkersComp`. If the product does not exist, a `BadIdentifierException` is thrown.
- `producerCodeId` - Reference to an existing PolicyCenter producer code. If the producer code does not exist, a `BadIdentifierException` is thrown.
- `policyNumber` - The unique ID number to assign to the imported policy. If the argument is `null` or references an existing PolicyCenter policy, the method generates a new and unique policy ID number.

- `policyPeriodData` - The policy period data to import. The `String` contents are parsed as XML that conforms to the PolicyCenter `PolicyPeriodModel` GX model. Load the GX model in Studio to view its structure.
- `changesToApply` - Not used in the base configuration. Custom implementations can use the argument to reference settings in the renewal policy period that differ from the legacy imported period.
- `parseOptions` - Not used in the base configuration. Custom implementations can use the argument for their own purposes.
- `basedOnEffectiveDate` - The effective date of the legacy imported policy period.
- `basedOnExpirationDate` - The expiration date of the legacy imported policy period. This date is also used as the effective date for the new imported renewal policy period. There is no gap in coverage between the legacy imported policy period and the renewal period.

If the `accountNumber`, `productCode`, or `producerCodeId` do not exist in PolicyCenter, the policy is not imported.

The imported `policyPeriodData` does not support attaching `Note` or `Document` objects to the renewal policy period. `Note` and `Document` objects can be attached to the renewal policy period after it has completed the renewal process.

The `basedOnEffectiveDate` value is passed to the Effective Time plugin's `getConversionRenewalEffectiveTime` method for processing.

The `basedOnExpirationDate` value is passed to the Effective Time plugin's `getConversionRenewalExpirationTime` method for processing.

The method creates a new policy on the specified account. Also, a new policy period is created and associated with the new policy. Finally, a renewal job is started for the new policy period. The renewal is in draft mode and is not bound. If the operation fails, the method throws a `DataConversionException`.

The method returns the renewal job's ID number as a `String`.

Policy renewal methods for billing systems

Billing systems use the `PolicyRenewalAPI` web service to send confirmation of renewals to PolicyCenter. Renewals can be configured to operate with or without renewal confirmation.

See also

- “Billing implications of renewals or rewrites” on page 521

Confirm a policy term

A billing system can notify PolicyCenter that an insured has confirmed a renewal. Confirmation usually takes the form of the insured submitting a payment for the renewal.

The `confirmTerm` method of the `PolicyRenewalAPI` web service confirms a policy term.

```
confirmTerm(policyNumber : String, termNumber : int, transactionId : String)
```

The method accepts the following arguments.

- `policyNumber` - Specifies the number of the confirmed policy
- `termNumber` - Specifies the number of the confirmed term
- `transactionId` - Unique transaction ID to use for the bundle

The `policyNumber` and `termNumber` arguments cannot be null. They must reference a bound and unarchived policy term.

The method has no return value.

The method calls the `IBillingSystemPlugin` method `updatePolicyPeriodConfirmed`. In the base configuration, the plugin method sets the specified policy term's `TermConfirmed` field to true.

Notify PolicyCenter of receipt of payment for renewal

The `notifyPaymentReceivedForRenewalOffer` method of the `PolicyRenewalAPI` web service notifies PolicyCenter that the billing system has received payment for a renewal offer.

```
notifyPaymentReceivedForRenewalOffer(offerID : String,
                                    selectedPaymentPlanCode : String,
                                    paymentAmount : MonetaryAmount,
                                    transactionId : String) : String
```

The method accepts the following arguments.

- **offerID** - Unique ID of the renewal offer. Cannot be `null`. The relevant policy period term must not be archived.
- **selectedPaymentPlanCode** - Billing payment plan code
- **paymentAmount** - Self explanatory
- **transactionId** - Unique transaction ID to use for the bundle

If the relevant policy period term is already bound, the method returns the period's policy number.

The method creates an activity and returns `null` if any of the following conditions occur.

- The policy is not in the correct state to accept the payment
- An applicable payment plan as specified by the `selectedPaymentPlanCode` and `paymentAmount` arguments could not be found

Otherwise, the method invokes the active workflow trigger `FinishIssueRenewal` and returns the period's policy number.

Archiving web services

To manipulate archiving from external systems, call the `ArchiveAPI` web service.

See also

- “Archiving integration” on page 617

Check if a policy term is archived

To check whether a policy term is archived, call the `ArchiveAPI` method `isArchived`. It takes a `String` policy number and an effective date. The method first determines which policy term is effective at the given date. The method returns `true` if the term is archived, otherwise `false`.

A term is considered archived if one or more of its policy periods is archived, even if not all are archived.

Restore a policy term

To request PolicyCenter restore a policy term, call the `ArchiveAPI` method `requestRestore`. The restore does not happen synchronously. In other words, the caller cannot assume the restore is complete when the API returns control to the caller. Instead, PolicyCenter simply adds this policy term to the set of policy terms to restore. Eventually, a batch process restores the policy term.

The `requestRestore` method accepts a `String` policy number and an effective date. The method first determines which policy term is effective at the given date. Next, it marks that policy term as a term to restore.

Suspend and resume archiving for a policy

PolicyCenter includes a feature to mark a policy as ineligible for additional archiving. This feature sometimes is known as suspending archiving. There are three methods on the `ArchiveAPI` web service that relate to the suspension of archiving. All three methods take a single argument, which is the policy number of the policy in question. The methods implement the following operations.

- Suspend archiving – From an external system, call the `ArchiveAPI` method `setDoNotArchive`.
- Resume archiving – From an external system, call the `ArchiveAPI` method `unsetDoNotArchive`. If there are terms that are already archived, resuming archiving does not restore any already-archived terms.
- Check a policy for suspended from archiving – From an external system, call the `ArchiveAPI` method `isDoNotArchive`. The method returns true if the policy is currently suspended from archiving, false otherwise.

The return result does not indicate whether the policy has any archived terms, only whether the policy is currently suspended from consideration of archiving.

Quote purging web services

To manipulate quote purging from external systems, call the `PurgeAPI` web service. The service provides methods for flagging whether to purge a policy or policy period.

See also

- *Application Guide*

Do not purge flag on a policy period

The `PurgeAPI` web service has methods related to the `DoNotDestroy` flag on a policy period. The methods accept the `PublicID` of the policy period. They perform the following operations.

- Check whether a policy period is excluded from purge – From an external system, call the `isDoNotDestroyPolicyPeriod` method to check the `DoNotDestroy` flag on a policy period.
- Flag a policy period as excluded from purge – From an external system, call the `setDoNotDestroyPolicyPeriod` method to set the `DoNotDestroyFlag` on a policy period to `true`.
- Flag a policy period as not excluded from purge – From an external system, call the `unsetDoNotDestroyPolicyPeriod` method to set the `DoNotDestroyFlag` on a policy period to `false`.

Part 3

Plugins

Overview of plugins

PolicyCenter plugins are classes that PolicyCenter invokes to perform an action or calculate a result at a specific time in its business logic. PolicyCenter 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 PolicyCenter 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.

From a technical perspective, PolicyCenter 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 only register 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 PolicyCenter 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, PolicyCenter 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 (Java with OSGi)	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 a OSGi bundle to reduce conflicts with other plugin code or PolicyCenter 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

<code>helpText</code>	Debug message information. Default is "".
<code>name</code>	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.
<code>required</code>	Boolean value specifying whether the parameter is required. Default is <code>false</code> .
<code>type</code>	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> • <code>String</code> • <code>StringArray</code> • <code>URL</code>

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 PolicyCenter 9.0.0 would include the characters pc900. 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 PolicyCenter 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.

PolicyCenter 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 PolicyCenter 10.0.0 includes pc1000. 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 PolicyCenter knows about it. The registry configures which implementation class is responsible for which plugin interface. If you correctly register your plugin implementation, PolicyCenter 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 PolicyCenter.

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 PolicyCenter 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, PolicyCenter 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.

To use plugin parameters, a plugin implementation must implement the `InitializablePlugin` interface in addition to the main plugin interface. If PolicyCenter detects that your plugin implementation implements `InitializablePlugin`, PolicyCenter 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 139

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 `PolicyCenter/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. PolicyCenter supports OSGi bundles only to implement a PolicyCenter 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 708
- “Using IntelliJ IDEA with OSGi editor to deploy an OSGi plugin” on page 709

Error handling in plugins

PolicyCenter 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")
    }
}
```

Enable or disable a plugin

Navigate to the Plugins Registry editor and select the relevant plugin. Clear or select the **Disable destination** check box to enable or disable the plugin.

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

- “Accessing entity data from Java” on page 702.

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.

PolicyCenter 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, PolicyCenter 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 138.

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 policy to evaluate to a specific `Policy` 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){
    e.printStackTrace()
    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 PolicyCenter 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 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 PolicyCenter 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 Policy 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 PolicyCenter 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.

PolicyCenter 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 PolicyCenter 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 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 PolicyCenter 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 PolicyCenter 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.pc.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 PolicyCenter 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 PolicyCenter 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 PolicyCenter 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 PolicyCenter plugins

The following tables summarize the plugin interfaces that PolicyCenter 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> <code>ClusterFastBroadcastTransportFactory</code> <code>ClusterUnicastTransportFactory</code>	Provides support for communication among nodes in a Guidewire PolicyCenter cluster. See <i>System Administration Guide</i> for more information.

Plugin Interface	Description
<code>IActivityEscalationPlugin</code>	Overrides the behavior of activity escalation instead of simply calling rule sets. See “Exception and escalation plugins” on page 266.
<code>IAddressAutocompletePlugin</code>	Configures how address automatic completion and fill-in operate. See “Automatic address completion and fill-in plugin” on page 271.
<code>IBaseURLBuilder</code>	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 269.
<code>IBatchCompletedNotification</code>	Launches custom actions after a work queue or batch process completes processing a batch of items.
<code>IEmailTemplateSource</code>	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.
<code>IEncryptionPlugin</code>	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. PolicyCenter does not provide any encryption algorithm in the product. PolicyCenter 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 251. You can register multiple implementation for this interface.
<code>IGroupExceptionPlugin</code>	Overrides the behavior of group exceptions instead of simply calling rule sets. See “Exception and escalation plugins” on page 266.
<code>InboundIntegrationStartablePlugin</code>	High performance inbound integrations, with support for multi-threaded processing of work items. See “Multi-threaded inbound integration” on page 283. You can register multiple implementation for this interface to communicate with multiple external systems.
<code>INoteTemplateSource</code>	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.
<code>IPhoneNormalizerPlugin</code>	Normalizes phone numbers that users enter through the application and that enter the database through data import. See “Phone number normalizer plugin” on page 271.
<code>IPreupdateHandler</code>	Implements your preupdate handling in plugin code rather than in rules. See “Preupdate handler plugin” on page 264.
<code>IProcessesPlugin</code>	Instantiates custom batch processing classes so they can be run on a schedule or on demand. See “Implementing <code>IProcessesPlugin</code> ” on page 661.
<code>IRestDispatchPlugin</code>	Processes REST API requests, including the following: <ul style="list-style-type: none"> • Preprocess incoming REST API requests • Rewrite outgoing API responses • Control how and what PolicyCenter logs for each API request
<code>IScriptHost</code>	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.
<code>IStratablePlugin</code>	Creates new plugins that immediately instantiate and run on server startup. See “Startable plugins” on page 275. You can register multiple implementation for this interface.
<code>ITestingClock</code>	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 PolicyCenter.

Plugin Interface	Description
	See “Testing clock plugin (for non-production servers only)” on page 273.
IUserExceptionPlugin	Overrides the behavior of user exceptions instead of simply calling rule sets. See “Exception and escalation plugins” on page 266.
IWorkItemPriorityPlugin	Calculates a the processing priority of a work item. See “Work item priority plugin” on page 266.
ManagementPlugin	The external management interface for PolicyCenter, which enables you to implement management systems, such as JMX and SNMP. See “Management integration” on page 259.

Plugins specific to Guidewire PolicyCenter

Plugin Interface	Description
AccountLocationPlugin	For account locations and policy locations, customizes how PolicyCenter performs the following operations: <ul style="list-style-type: none"> Determine that two locations are the same Clone a location See “Location plugin” on page 159.
ConversionOnRenewal	Configures actions to perform if a renewal fails during converting. See “Conversion on renewal plugin” on page 155.
IAccountPlugin	Manipulates accounts. See “Account plugin” on page 153.
IArchivingSource	Stores and retrieves archived policies from an external backing source. See “Archiving integration” on page 617.
IAuditSchedulePatternSelectorPlugin	Customizes how PolicyCenter chooses the audit schedule pattern for cancellation and expiration. See “Audit schedule selector plugin” on page 154.
IBillingSummaryPlugin	Generates billing summaries for the billing summary screen. See “Implementing the billing summary plugin” on page 545
IBillingSystemPlugin	Interacts with an external billing system for actions that PolicyCenter initiates. See “Implementing the billing system plugin” on page 533
IClaimSearchPlugin	Searches for claims. See “Claim search from PolicyCenter” on page 479.
IEffectiveTimePlugin	Determines the time of day (since midnight) for a job. See “Effective time plugin” on page 155.
IETLProductModelLoaderPlugin	Extracts product model data from the running PolicyCenter server, and stores the information in ETL product model tables in the PolicyCenter database. See “ETL Product Model Loader plugin” on page 157.
IExchangeRateSet	Internal only. Never use or implement this plugin.
IFXRatePlugin	Handles exchange rate conversion.
IImpactTestingPlugin	Only for customers who license Guidewire Rating Management. This plugin configures rating routine impact testing.
IJobCreation	Creates a job process. See “Job process creation plugin” on page 158
IJobProcessCreationPlugin	Creates the right job process subtype. See “Job process creation plugin” on page 158.

Plugin Interface	Description
ILossHistoryPlugin	Handles loss history summaries for a policy. See “Loss history plugin” on page 160.
IMotorVehicleRecordPlugin	Generates a request to the motor vehicle record (MVR) provider and returns the MVR data. See “Motor vehicle record (MVR) plugin” on page 160.
INotificationPlugin	Customizes the minimum lead time and the maximum lead time for different types of notifications. See “Notification plugin” on page 162.
IPolicyEvaluationPlugin	Customizes how PolicyCenter adds underwriter issues (<code>UWIssue</code> objects) on a policy period. See “Policy evaluation plugin” on page 165.
IPolicyNumGenPlugin	Generates policy numbers of two types: core policy numbers and current policy revision numbers. See “Policy number generator plugin” on page 167.
IPolicyPaymentPlugin	Returns the filtered reporting plans based on the policy. See “Policy payment plugin” on page 167.
IPolicyPeriodDiffPlugin	Customizes how PolicyCenter generates and filters difference items that represent comparing two policies for policy changes, multi-version policy comparisons, and other contexts. See “Policy difference and comparison customization” on page 457.
IPolicyPeriodPlugin	<p>Various important policy period customizations, including the following:</p> <ul style="list-style-type: none"> • Calculating the period end date from a <code>TermType</code> • Calculating a <code>TermType</code> from period start and end dates • Dynamically setting the initial wizard step in the job wizard • Creating a job process • Prorating bases from cancellation • Specifying types to omit while copying a <code>PolicyPeriod</code> • Customizing behavior after creating draft branch in new period • Customizing behavior before promoting a branch • Customizing behavior after handing a preemption <p>See “Policy period plugin” on page 168.</p>
IPolicyPlugin	Informs PolicyCenter whether it is OK to start various jobs, based on dynamic calculations on the policy. See “Policy plugin” on page 171.
IPolicyTermPlugin	Calculates values related to policy terms. See “Policy term plugin” on page 174
RateBookPreloadPlugin	<p>Only for customers who license Guidewire Rating Management.</p> <p>The preloading Rating Management components feature and precompiling rate routines JAR feature use this plugin. See <i>Configuration Guide</i>.</p>
IRateQueryLookupPlugin	<p>Only for customers who license Guidewire Rating Management.</p> <p>This plugin enables you to customize the rate query fail-over logic in PolicyCenter.</p>
IRateRoutinePlugin	Only for customers who license Guidewire Rating Management.

Plugin Interface	Description
<code>IRatingPlugin</code>	This plugin configures processing of rate routines.
<code>IReferenceDatePlugin</code>	This is the main plugin interface that defines the interaction between the application and a rating engine. See “Rating integration” on page 379.
<code>IRenewalPlugin</code>	Evaluates the type of date to use to calculate the reference date on a given object. See “Reference date plugin” on page 181.
<code>IUWCompanyPlugin</code>	Handles renewals. See “Renewal plugin” on page 181.
<code>IVinPlugin</code>	Finds all the underwriting companies that are available for the jurisdictions in the set. See “Underwriting company plugin” on page 182.
<code>JobNumberGenPlugin</code>	Allows an external system to provide information about vehicles. See “Vehicle identification number plugin” on page 271.
<code>PaymentGatewayConfigPlugin</code>	Generates a new, unique job number. See “Policy number generator plugin” on page 167.
<code>PaymentGatewayPlugin</code>	Configures the payment gateway that PolicyCenter uses to communicate with an electronic payment service. See “Payment gateway configuration plugin” on page 163.
<code>PCPurgePlugin</code>	Communicates with an electronic payment service. See “Payment gateway plugin” on page 164.
<code>ProrationPlugin</code>	Modifies the behavior of quote purging. Quote purging removes from the database jobs not resulting in bound policies and alternate policy periods created through multi-version quoting and side-by-side quoting. Quote purging also removes orphaned policy periods, which are policy periods not associated with a job. See “Quote purging plugin” on page 178.
<code>QuotingDataPlugin</code>	Prorates a value. This is used by two places in PolicyCenter: <ul style="list-style-type: none"> • Generate transaction calculations • Rating integration See “Proration plugin” on page 176.
<code>WorksheetExtractPlugin</code>	Provides methods for saving quoting data to an external database.
<code>WorksheetPurgePlugin</code>	Only for customers who license Guidewire Rating Management. This plugin identifies which rating worksheet to extract and extracts the worksheet data to files.
<code>ContactSystemPlugin</code>	Only for customers who license Guidewire Rating Management. This plugin configures how PolicyCenter purges rating worksheets.

Plugins for managing PolicyCenter contacts

Plugin Interface	Description
<code>IAccountContactPlugin</code>	Searches for contacts and retrieve contacts from an external system. See “Integrating with a contact management system” on page 565.
<code>ContactSystemPlugin</code>	Configures how to copy properties between account contacts. See “Account contact plugin” on page 573.

Plugin Interface	Description
IAccountContactRolePlugin	Configures how to copy properties between account contact roles and copy pending updates. See “Account contact role plugin” on page 573.
IAccountSyncable	Customizes how PolicyCenter synchronizes contact with accounts. For more information. See “Synchronizing contacts with accounts” on page 572.
IAddressBookAdapter	Internal only. Never use or implement this plugin.
IContactConfigPlugin	Configures contacts, such as the address contact roles that are available and getting the display name for an account contact role type. See “Configuring how PolicyCenter handles contacts” on page 571.
IContactSearchAdapter	Internal only. Never use or implement this plugin.
IGeocodePlugin	Geocoding support in PolicyCenter. See “Geographic data integration” on page 241.
OfficialIdToTaxIdMappingPlugin	Determines whether PolicyCenter treats an official ID type as a tax ID for contacts. See “Official IDs mapped to tax IDs plugin” on page 272.

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 PolicyCenter. 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 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 58.

Plugins for managing document content and metadata

Plugin Interface	Description
IDocumentContentSource	Provides access to a remote document repository for storage and retrieval operations. 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.
IDocumentMetadataSource	Stores metadata associated with a document, typically in a remote document management system.

Plugin Interface	Description
	The example <code>IDocumentDataSource</code> 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.

Plugins for managing document production

Plugin Interface	Description
<code>IDocumentProduction</code>	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 217.
<code>IDocumentTemplateSerializer</code>	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. Use the built-in version of this plugin using the “Plugins registry.” In general, it is best not implement your own version.
<code>IDocumentTemplateSource</code>	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
<code>MessageAfterSend</code>	Optional post-send processing of messages. You can register multiple implementation for this interface to communicate with multiple external systems.
<code>MessageBeforeSend</code>	Optional pre-processing of messages, primarily to transform the payload and return a <code>String</code> 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. You can register multiple implementation for this interface to communicate with multiple external systems.
<code>MessageReply</code>	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 synchronously, do not implement this plugin. Instead, implement only the transport plugin and acknowledge each message immediately after it sends the message. You can register multiple implementation for this interface to communicate with multiple external systems.
<code>MessageRequest</code>	Optional pre-processing of messages, and optional post-send processing. The <code>MessageRequest</code> interface perform roles of both the related plugins <code>MessageBeforeSend</code> and <code>MessageAfterSend</code> . You can register multiple implementation for this interface to communicate with multiple external systems.
<code>MessageTransport</code>	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. You can register multiple implementation for this interface to communicate with multiple external systems.

Plugins for REST API Framework

Plugin interface	Description
IRestDispatchPlugin	Optional plugin interface to do the following: <ul style="list-style-type: none">• Preprocess incoming REST API requests• Rewrite outgoing API requests• Control how and what PolicyCenter logs for each API request See the <i>REST API Framework</i> documentation for more information.

Account and policy plugins

Plugins are software modules that PolicyCenter calls to perform an action or calculate a result. This topic details plugins related mainly to PolicyCenter accounts, policies, and policy-related jobs.

Account plugin

You can use the account plugin (`IAccountPlugin`) to customize how PolicyCenter manages accounts.

Performing account name clearance

Name clearance is part of an insurance application in which the insurance insurer determines whether this client already has insurance with the insurer or is represented by a different producer. If represented by another producer, the account is reserved for the other producer. Reservation is performed by line of business, so a client represented by one producer for one line of business could be represented by another producer for another type of policy. If the name is reserved, then the producer needs to be informed of the conflict. If not, the account is now reserved for this producer and the process can proceed.

The plugin's `performNameClearance` method performs name clearance on the given account and producer in the form of an `Account` and a `ProducerSelection`. It returns the available products and sets the status appropriately based on the product model configuration. The status on a product might be something other than available, for example that the product is risk reserved or inapplicable.

Deleting an account

PolicyCenter calls this plugin's `freezeAccount` method to signal that the passed-in `Account` entity is frozen and about to be deleted. If you need to do any special action before deletion, do it in this method. For example, if you have linked data through foreign keys on the `Account` entity, you might need to delete that data also.

PolicyCenter only calls this if PolicyCenter is not the true system of record for the account and the account is about to be deleted from the account system of record. This is also called after withdrawing accounts. PolicyCenter does not allow deletion of accounts if PolicyCenter is the account system of record.

Generating a new account number

Generate a new account number in the `generateAccountNumber` method. It takes an `Account` number and must return a unique ID for the account. For more information, see "Creating unique numbers in a sequence" on page 144.

Is account editable?

Implement the `isEditable` method in this plugin to indicate whether an account is editable. It takes an `Account` object and returns a `boolean`.

The default behavior is that an account is editable, but you can customize this behavior. For example, if PolicyCenter is not the system of record for the account, you might choose to always make the account non-editable.

Checking whether risk is reserved

Implement the `isRiskReserved` method to customize how PolicyCenter checks whether the given `PolicyPeriod` is already risk reserved.

Customizing behavior after merging accounts

After PolicyCenter merges two accounts, calls the `mergeAccounts` method in this plugin so you can customize the application logic. There is no required behavior, but if you have data model extension properties, you might need to merge or copy data to the merged account. If you have related foreign key fields to other data, you might need to merge or delete that data also.

PolicyCenter calls this method after the two accounts were merged into a single entity but the other (non-merged) account entity has not yet been retired. The first parameter to the method is `Account` that is soon to be retired. The second parameter is the destination merged account.

Merging two accounts is not enabled in the built-in PolicyCenter user interface, but can be triggered using the `AccountAPI` web service `mergeAccounts`.

Populating account summaries with extension properties

PolicyCenter has an entity called `AccountSummary` that it uses in several places in the user interface to summarize an account. The application copies the standard built-in properties automatically.

There is no required behavior. However, if you have data model extension properties in the `Account` that you want in the summary, copy them from the `Account` to the `AccountSummary`. Implement this action in the account plugin's `populateAccountSummary` method.

Transferring policies from one account to another account

After PolicyCenter transfers policies from one account to another, you can customize the application logic.

There is no required behavior, but if you have any data model extension properties and possibly additional foreign key references, you might need to customize this logic. Implement the `transferPolicies` method, with the following signature.

```
void transferPolicies(Policy[] policies, Account fromAccount, Account toAccount)
```

Transferring policies from one account to another is not enabled in the built-in PolicyCenter user interface, but you can trigger it using the `AccountAPI` web service method `transferPolicies`. For more information, see “Account web services” on page 114.

Audit schedule selector plugin

You can customize how PolicyCenter chooses the audit schedule pattern for cancellation and expiration by implementing the `IAuditSchedulePatternSelectorPlugin` plugin interface. You define (and edit) the audit schedule patterns in Product Designer as part of the product model.

The built-in implementation is provided in `AuditSchedulePatternSelectorPlugin.gs`, which you can modify or use as the basis for your own version.

PolicyCenter includes several schedules in the built-in implementation. The built-in plugin implementation references several of these.

There are two methods that you must implement.

- `selectFinalAuditSchedulePatternForCancellation` – Returns the cancellation audit schedule pattern to use for final audits. It takes a `PolicyPeriod` and returns an `AuditSchedulePattern` object.
- `selectFinalAuditSchedulePatternForExpiredPolicy` – Returns the expiration audit schedule pattern to use for final audits. It takes a `PolicyPeriod` and returns an `AuditSchedulePattern` object.

In the built-in implementation, these methods ignore the `PolicyPeriod`. Instead, the methods simply iterate across all schedule patterns to find a pattern that matches a specific pattern code.

```
class AuditSchedulePatternSelectorPlugin implements IAuditSchedulePatternSelectorPlugin {  
  
    override function selectFinalAuditSchedulePatternForCancellation(period : PolicyPeriod) :  
        AuditSchedulePattern {  
            return AuditSchedulePatternLookup.getAll().firstWhere(\ f -> f.Code == "CancellationPhone")  
        }  
  
    override function selectFinalAuditSchedulePatternForExpiredPolicy(period : PolicyPeriod) :  
        AuditSchedulePattern {  
            return AuditSchedulePatternLookup.getAll().firstWhere(\ f -> f.Code == "ExpirationPhysical")  
        }  
}
```

Conversion on renewal plugin

To configure actions to perform if a renewal fails during converting, implement the `ConversionOnRenewal` plugin. There is only a single method, called `conversionOnRenewalFailed`. Before calling this plugin method, PolicyCenter unlocks the `PolicyPeriod` and sets its status to `draft` so that it is editable.

The plugin can perform actions that improve the chances of the renewal conversion succeeding.

PolicyCenter provides a default plugin implementation called `gw.plugin.job.impl.ConversionOnRenewalPluginImpl`. The class implements the following default behavior.

- Set the policy period status to `TC_NEW`, which allows eventual purging.
- Reassign a new policy number to each policy period, to reduce chance of conflict with existing policies.

If you want a different behavior, re-implement this plugin interface.

The `conversionOnRenewalFailed` method has two parameters: A renewal period (`Renewal`) and the original policy period for the renewal (`PolicyPeriod`).

The method must return the original `Renewal` object passed to the method. Do not create and return a new `Renewal` object.

Effective time plugin

Certain `PolicyPeriod` properties that denote effective dates and expiration dates are treated as `datetime` properties with millisecond resolution. You can set the effective time (the clock time on that day) along with the effective date. You can choose to have all effective dates always be the same time on any day, such as midnight or 12:01 AM on that day. However, you can also choose to use more complex rules, and even vary the rules based on the type of job or the jurisdiction if rules vary by state. PolicyCenter provides a plugin interface that lets you set the effective time of day for various jobs. Modify the built-in version or implement your own implementation of the `IEffectiveTimePlugin` plugin.

After a job starts, PolicyCenter calls this plugin to set the default effective and expiration times on the job's associated `PolicyPeriod` entity. The return value from each of this interface's methods is a `Date` object with its time component (`HH:mm:ss`) set to the desired time. The day component of the `Date` is ignored. As a convenience, you can return `null` to set the time component to midnight.

The plugin is called at varying points during job setup, according to the type of job. The main reason for this difference is to allow you to customize the user interface to allow effective/expiration time settings to be manually overridden. The following table compares the types of jobs, at what time the application calls the plugin, and why.

What time PolicyCenter calls the Effective Time plugin	For which job types	Description
Before job is created	Policy Change, Cancellation. These are jobs that mainly affect a period's <code>EditEffectiveDate</code> property.	The only place where <code>EditEffectiveDate</code> can be edited is on the <code>StartPolicyChange</code> or <code>StartCancellation</code> page, so the plugin must be called within the PCF file before the job is started.
After job is created	Submission, Reinstatement, Rewrite, and Renewal. These are jobs that mainly affect a period's <code>PeriodStart</code> and <code>PeriodEnd</code> properties.	The effective/expiration time settings on <code>PeriodStart</code> and <code>PeriodEnd</code> can be made editable by simply customizing the <code>PolicyInfo</code> page, which is not shown until after the job is started.
Never	Issuance, Audit	This plugin is not called for the Issuance job because it is logically seen as a continuation/completion of a Submission job. Consequently, its <code>PolicyPeriod</code> simply inherits the dates set by the Submission job. Similarly, there are no hooks for the Audit job, since it simply uses the dates of the <code>PolicyPeriod</code> being audited.

Each method is responsible for determining either an effective time or expiration time for a date given the date and the job type. The methods for determining expiration time also take an `effectiveDateTime` parameter, for cases in which the expiration time depends on the period's `PeriodStart`. Each method takes a `PolicyPeriod` entity parameter. Use this parameter to access other information, such as state, line of business, or detecting other contractual periods on this policy. Finally, each method has a strongly-typed job parameter, which can be used to access information specific to the associated job type.

The following table lists the required methods and their purpose. Each method returns a `java.util.Date` object.

Method	Description
<code>getCancellationEffectiveTime</code>	Gets effective time for a cancellation job.
<code>getConversionRenewalEffectiveTime</code>	Gets a renewal policy period's effective date for an imported legacy policy.
<code>getConversionRenewalExpirationTime</code>	Gets a renewal policy period's expiration date for an imported legacy policy. Includes an argument for the renewal policy period's effective date, as returned by <code>getConversionRenewalEffectiveDate</code> .
<code>getPolicyChangeEffectiveTime</code>	Gets effective time for a policy change job.
<code>getReinstatementEffectiveTime</code>	Gets effective time for a reinstatement job.
<code>getReinstatementExpirationTime</code>	Gets expiration time for a reinstatement job. Includes an argument for the reinstated policy period's effective date, as returned by <code>getReinstatementEffectiveTime</code> .
<code>getRenewalEffectiveTime</code>	Gets effective time for a renewal job.
<code>getRenewalExpirationTime</code>	Gets expiration time for a renewal job. Includes an argument for the renewal policy period's effective date, as returned by <code>getRenewalEffectiveTime</code> .
<code>getRewriteEffectiveTime</code>	Gets effective time for a rewrite job.
<code>getRewriteExpirationTime</code>	Gets expiration time for a rewrite job. Includes an argument for the rewritten policy period's effective date, as returned by <code>getRewriteEffectiveTime</code> .
<code>getRewriteNewAccountEffectiveTime</code>	Gets a policy period's effective date for a policy that was rewritten to a new account.
<code>getRewriteNewAccountExpirationTime</code>	Gets a policy period's expiration date for a policy that was rewritten to a new account. Includes an argument for the policy period's effective date, as returned by <code>getRewriteNewAccountEffectiveTime</code> .
<code>getSubmissionEffectiveTime</code>	Gets effective time for a submission job.

Method	Description
getSubmissionExpirationTime	Gets expiration time for a submission job. Includes an argument for the policy period's effective date, as returned by getSubmissionEffectiveTime.

PolicyCenter includes a built-in implementation of this interface with default behavior. The following table lists effective times for a new job and the key values on the **PolicyPeriod** entity for that job after PolicyCenter calls this plugin. The shaded cells are the effective times set explicitly by this plugin and the other cells indicate unchanged values or dependence on other properties.

Job	Value of PeriodStart	Value of PeriodEnd	Value of EditEffectiveDate	Value of CancellationDate
Submission	12:01 AM (changed by plugin)	12:01 AM (changed by plugin)	Same as this period's PeriodStart	null
Policy Change	Unchanged	Unchanged	12:01 AM	Unchanged
Cancellation	Unchanged	Unchanged	If canceled flat, same as canceled period's PeriodStart. Otherwise, 12:01 AM (changed by plugin).	Same as this period's EditEffectiveDate
Reinstatement	Unchanged	Same as based-on period's PeriodEnd (changed by plugin).	Same as based-on period's CancellationDate (changed by plugin)	null
Rewrite	Same as based-on period's CancellationDate (changed by plugin).	If new-term rewrite, 12:01 AM. Otherwise, such as full-term or remainder-of-term rewrite, same as based-on period's PeriodEnd (changed by plugin).	Same as this period's PeriodStart	null
Renewal	Same as previous period's PeriodEnd (changed by plugin).	12:01 AM (changed by plugin)	Same as this period's PeriodStart	null

ETL Product Model Loader plugin

The ETL Product Model Loader plugin extracts product model data from the running PolicyCenter server, and stores the information in ETL product model tables in the PolicyCenter database. Policy data combined with the ETL product model data can be extracted, transformed, and loaded into a data warehouse or data store for analysis or reporting.

When the PolicyCenter server starts, it runs the ETL Product Model Loader plugin. PolicyCenter provides two implementations of this plugin interface (**IETLProductModelLoaderPlugin**).

- **ETLProductModelLoaderPlugin** – Operational plugin implementation that is enabled and registered in the base configuration.
- **ETLProductModelLoaderEmptyPlugin** – Non-operational plugin implementation.

Only one implementation can be enabled at a time.

See also

- *Product Model Guide*

Plugin implementation

In the base configuration, the ETL Product Model Loader plugin implementation is registered and enabled in the plugins registry. This plugin implementation first removes data from the ETL product model database tables. Then

the plugin extracts the PolicyCenter product model to ETL entities and writes these entities to the ETL product model database tables.

If you update the product model in Product Designer, first synchronize the product model from Product Designer to propagate your changes to PolicyCenter. Then restart PolicyCenter to run the Product Model Loader plugin. The plugin repopulates the ETL database tables with the current product model.

If you add policy lines or add clauses and coverage terms to existing policy lines, the Product Model Loader creates ETL product model tables for these additions. You can also extend the Product Model Loader to access other product model data, including product offerings or questions in question sets. These changes may require modifying the ETL entities.

The Gosu code for the Product Model Loader plugin is in `ETLProductModelLoaderPlugin.gs` in the `gw.plugin.etlprodmodloader.impl` package. In the `start` method, the plugin first clears the ETL database tables and entities by calling the `deleteModel` method. Then the plugin populates the ETL database tables and entities by calling the `loadModel` method.

The `factory` subpackage contains creator classes for each ETL product model entity type. For example, the `ETLClausePatternCreator.gs` class contains code to create `ETLClausePattern` entity instances.

Non-operational plugin implementation

To effectively disable the ETL Product Model Loader plugin, register and enable the Non-operational ETL Product Model Loader plugin implementation. This implementation does not populate the ETL database tables. Nor does the implementation clear the ETL database tables.

The code for the Non-operational ETL Product Model Loader plugin implementation is in `ETLProductModelLoaderEmptyPlugin` in the `gw.plugin.etlprodmodloader.impl` package.

In the registry for this plugin (`IETLProductModelLoaderPlugin.gwp`), perform the following operations.

- Register and enable the `ETLProductModelLoaderEmptyPlugin` plugin implementation.
- Remove the `ETLProductModelLoaderPlugin` plugin implementation from the registry.

Job number generator plugin

The `IJobNumberGenPlugin` job number generator plugin interface is responsible for generating a job number. The base configuration plugin implementation in the Gosu `JobNumberGenPlugin` class is for demonstration purposes only. You must replace the built-in implementation with your own version of the plugin before moving your system into production.

The interface method `genNewJobNumber` takes an array of parameter strings and returns a non-null unique identifier string for the job.

The input parameter strings do not have a Guidewire-defined meaning, and you can customize the use of these parameters in user interface code that calls this plugin. You can change the Gosu code to pass values other than the default value of `null`.

The `JobNumberGenPlugin` plugin is the built-in implementation of this interface. This plugin checks for job numbers that already exist, and generates a new random value. The value is randomly generated using the procedure described below.

- The first five digits increment a counter. This counter is reset at server restart.
- The last five digits are random.

For example, in a single server startup, job numbers increase. The first job is `00000<random5>`, the second is `00001<random5>`, the third is `00002<random5>`, and so on. When the server restarts, the counter resets to zero, and therefore job numbers do not reflect creation time. When the counter reaches the limit of five digits, the counter increases to six digits. Therefore, the plugin returns 11 digit job numbers.

Job process creation plugin

The `JobProcessCreationPlugin` job process creation plugin creates a new job process for a policy period entity based on its job subtype. Its one method, called `createJobProcess`, takes a `PolicyPeriod` entity and returns an

instance of a `JobProcess`. For example, for a policy change job, this plugin must create a `PolicyChangeProcess` object. The included version of this plugin creates job processes for all built-in job subtypes.

The plugin method is important if you created subtypes of one or more of the built-in job process classes. If that is the case, you must detect your new subtypes in this method. Checking the value of `Job` property on the `PolicyPeriod` and check its subtype. Next, instantiate your own job process subtype and return it.

Use the code of the included version of this plugin as a template for your version.

```
class JobProcessCreationPlugin implements IJobProcessCreationPlugin {  
    ...  
    override function createJobProcess(period: PolicyPeriod): IJobProcess {  
        switch (period.Job.Subtype) {  
            case "Audit": return new AuditProcess(period)  
            case "Cancellation" : return new CancellationProcess(period)  
            ...  
        }  
    }  
}
```

See also

- *Configuration Guide*

Location plugin

You can customize how PolicyCenter treats account locations and policy locations using the `AccountLocationPlugin` location plugin. The base configuration of the plugin is implemented in the Gosu `AccountLocationPluginImpl` class.

The basic tasks you can customize are described below.

- Customize how two location records are compared to see if they represent the same location or two different locations.
- Customize how to copy your data model extension properties during cloning of a location entity.

If you want to add custom code after creating a primary location or before deleting a location, modify each policy line's methods called `onPrimaryLocationCreation` and `preLocationDelete`. Similarly, to define whether a location is safe to delete, modify each policy line's `canSafelyDeleteLocation` method.

Compare locations

Your `areAccountLocationsEquivalent` method must return `true` if two `AccountLocation` entities passed as parameters are effectively equal.

The behavior in the built-in implementation returns `true` if and only if the following properties match: `AddressLine1`, `AddressLine2`, `AddressLine3`, `City`, `State`, and `PostalCode`.

If you have data model extension properties on your locations that are important in the comparison, customize this logic as needed.

Location cloning

If you have data model extension properties or arrays on an `AccountLocation` or one of its subtypes, copy them if an account location entity clones to another account location. Implement this plugin's `cloneExtensions` method to copy these properties, including any foreign keys or array information. The built-in implementation does nothing.

```
override function cloneExtensions(oldLocation: AccountLocation, newLocation: AccountLocation){  
}
```

It is critical that you not make any changes to the related `PolicyPeriod` entities, `Account` entities, or any other entity within this method. PolicyCenter calls this method while binding a `PolicyPeriod` entity. If you modify any entity other than the new location contact, the policy could become out of sync and create serious problems later on.

Only copy your data model extensions from the old account contact to the new one and do nothing else in this method. In particular, this method must not make changes to a related `PolicyPeriod`, `Account`, or other entity or else serious problems can occur.

Loss history plugin

The `ILossHistoryPlugin` interface retrieves the loss history financial information for a particular account or policy period.

The base configuration implements the interface in the `LossHistoryPlugin` class. The class methods retrieve financial data stored in PolicyCenter. If desired, the methods can be customized to retrieve data stored in an external system.

Retrieve loss history for an account

The `getLossFinancialsForAccount` method retrieves PolicyCenter financial information for all policies in a specified account.

```
getLossFinancialsForAccount(accountNumber : String) : LossFinancials[]
```

The `accountNumber` argument specifies the relevant account.

The method returns an array of `LossFinancials` objects that contain financial information for the individual policies in the account. Each object contains financial data for a policy's latest policy period.

The plugin method is invoked whenever PolicyCenter calls an `Account` entity's `getLossFinancials` method to retrieve the account's financial information.

Retrieve loss history for a policy period

The `getLossFinancialsForPolicy` method retrieves PolicyCenter financial information for a specified policy period.

```
getLossFinancialsForPolicy(policyNumber : String, periodAsOfDate : Date) : LossFinancials
```

The `policyNumber` argument specifies the relevant policy. The `periodAsOfDate` argument specifies an effective date used to identify the desired policy period.

The method returns a `LossFinancials` object that contains financial information for the policy period in effect at the specified `periodAsOfDate`.

The plugin method is invoked whenever PolicyCenter calls a `PolicyPeriod` entity's `getLossFinancials` method to retrieve the period's financial information.

Motor vehicle record (MVR) plugin

PolicyCenter provides the `IMotorVehicleRecordPlugin` to request a motor vehicle record (MVR) from an external MVR service provider. The base configuration of PolicyCenter uses a built-in implementation, `DemoMotorVehicleRecordPlugin.gs`. To edit the registry for the `IMotorVehicleRecordPlugin`, in Studio, navigate to `configuration`→`config`→`Plugins`→`registry`, and open the `IMotorVehicleRecordPlugin.gwp` file.

Built-in implementation of the motor vehicle record plugin

The base configuration of PolicyCenter provides a built-in demonstration implementation of the Motor Vehicle Record plugin, `DemoMotorVehicleRecordPlugin.gs`. This plugin implementation does not integrate with a specific MVR service provider. Instead, it simulates the reception of MVR reports for U.S. drivers.

In the base configuration, the `IMotorVehicleRecordPlugin` is enabled and specifies its implementation class as `gw.plugin.motorvehiclerecord.DemoMotorVehicleRecordPlugin`. Use the code in this demonstration class as an example for writing your own motor vehicle record plugin implementation.

Writing a motor vehicle record plugin

To implement requests to a real motor vehicle record (MVR) provider, write your own Gosu class that implements the `IMotorVehicleRecordPlugin` interface. Your implementation is responsible for contacting an MVR service provider and returning MVR data to PolicyCenter. It is not responsible for determining whether PolicyCenter already has a report for the driver in its MVR repository or whether the report in the repository is stale.

Data used by the motor vehicle record plugin

The methods defined by the `IMotorVehicleRecordPlugin` interface and their internal implementations use these data-type interfaces defined in the `gw.api.motorvehicleresult` package.

- `IMVROrder` – Information about an MVR order, such as internal or provider request IDs and order status
- `IMVRSUBJECT` – Header information about an MVR order, including the search criteria for a driver
- `IMVRSearchCriteria` – Search criteria for an MVR order, such as the driver's name and date of birth
- `IMVRData` – Incidents and licenses on a motor vehicle report
- `IMVRLicense` – Details of a license on a motor vehicle report
- `IMVRIncident` – Details of an incident on a motor vehicle report

Methods on the motor vehicle record plugin

The `IMotorVehicleRecordPlugin` interface defines three methods.

Method	Description
<code>orderMVR</code>	Sends an array of orders to the MVR service provider to request reports for specific drivers
<code>getMVROrderResponse</code>	Sends an array of orders to the MVR service provider asking which reports are ready
<code>getMVRDetails</code>	Sends an array of orders to the MVR service to obtain reports that are ready

Each method accept an array of `IMVRSUBJECT` instances as its sole input parameter.

How PolicyCenter uses motor vehicle records

A motor vehicle record (MVR) documents a driver's driving history. The MVR report contains information such as identifying data, license status, convictions, traffic violations, accidents, license suspensions, and revocations. In the U.S., the information in this report usually comes from the Department of Motor Vehicles (DMV) for each jurisdiction. The information in the report can vary by jurisdiction. In the U.S., most service providers provide MVR data for all jurisdictions, so you need to integrate only with a single service provider.

An insurer uses MVR reports to evaluate risks associated with drivers. Violations are assigned point values, with more severe violations having a higher point value. A high MVR point total indicates a high risk driver and can result in higher policy premiums. Agents who use PolicyCenter click **Retrieve MVR** on the **Drivers** screen of a personal auto job to start a `ProcessMVRsWF` workflow. This workflow calls the `IMotorVehicleRecordPlugin` to retrieve MVR reports for drivers submitted by agents.

Since there is a cost associated with retrieving reports from external MVR service providers, PolicyCenter stores the retrieved MVR data to minimize future lookups. PolicyCenter requests the MVR report from an external service provider only if PolicyCenter does not have a copy or if the report is considered to be stale.

The retrieved MVR report is maintained in a separate MVR repository that is independent of the driver's account or policy. This allows an MVR report to be reused for a driver associated with multiple accounts or policies. This also provides flexibility as to when to update an MVR report on in-force policies. In the base configuration, the MVR report for an in-force policy is updated at renewal or during a policy change.

See Also

- *Application Guide*

Notification plugin

You can customize notification dates for a variety of types of notifications. Implement the `INotificationPlugin` plugin or modify the built-in implementation of this plugin. This plugin is responsible for determining the minimum lead time and maximum lead time for different types of notifications. The law frequently specifies these values, so define and calculate these numbers carefully.

For the methods in this plugin, the effective date is a method parameter. It specifies the date at which the action occurs. For example, suppose the lead time is 10 days if the effective date is prior to June 5, and 15 days if the effective date is June 5 or later. The plugin must return the value 10 for effective dates prior to June 5 and 15 for effective dates on or after June 5. This effective date argument is never `null`.

Another parameter is the line to jurisdictions mapping, using generics syntax this is `Map<PolicyLinePattern, State[]>`. This maps from a line of business to an array of jurisdictions. For example, imagine a commercial package policy with two lines: general liability and commercial property. The general liability line has only one jurisdiction (California) and the commercial property line has two jurisdictions (California and Arizona). In this case, the passed map contains two mappings.

- One mapping from general liability to an array containing the single value of California
- One mapping from commercial property to an array containing the values California and Arizona

Large commercial policies likely have much large mappings that could contain dozens of jurisdictions. This method parameter is never `null` and never empty. None of the mappings contain empty arrays of jurisdictions.

Refer to the built-in implementation of this plugin for a general pattern to use. Also, the built-in implementation of this plugin (`gw.plugin.notification.impl.NotificationPlugin`) has some utility functions that you can use to simplify your code. Specifically, look at the code for the method `forAllLinesAndJurisdictions`.

The built-in implementation of this plugin calculates the results based on looking up the input values in the system table `NotificationConfig` (controlled by the file `notificationconfigs.xml`).

If you write your own version of this plugin, you can use the notification system table or use completely different logic to calculate the lead time dates.

Notification by action type (minimum)

The first minimum lead time method has the following method signature.

```
int getMinimumLeadTime(Date effDate, Map<PolicyLinePattern, State[]> lineToJurisdictions,
                      NotificationActionType actionType)
```

Notice that one of the parameters is a `java.util.Map` that maps a policy line pattern to an array of states. Using Gosu generics syntax, this is `Map<PolicyLinePattern, State[]>`. You must iterate across all the policy line patterns and the matching list of states and calculate the minimum lead time for all combinations of line of business and jurisdiction. Use the `actionType` typecode and effective date as appropriate. Your plugin method must return a value that represents the minimum lead time as a number of number of days.

The action type is an `NotificationActionType` typecode to indicate the type of notification for which PolicyCenter requests a date. A `NotificationActionType` includes typecodes such as those listed below.

- `fraudcancel` – Notification requirements in days for cancellation due to fraud
- `rateincrease` – Notification requirements in days for rate increases

The list of typecodes in this list is extensive. Refer to the typelist in Studio or the *Data Dictionary* for the complete list.

Notification by action type (maximum)

There is another method with identical method signature as the previously-mentioned method except the name is `getMaximumLeadTime`. That method must calculate the maximum lead time (not the minimum) for all combinations of line of business and jurisdiction.

```
int getMaximumLeadTime(Date effDate, Map<PolicyLinePattern, State[]> lineToJurisdictions,
                      NotificationActionType actionType)
```

Notification by category (minimum)

Another variant of the `getMinimumLeadTime` method has the following method signature.

```
int getMaximumLeadTime(Date effDate, Map<PolicyLinePattern, State[]> lineToJurisdictions,
                       NotificationCategory category) throws RemoteException;
```

As you can see, it takes a notification category (`NotificationCategory`) instead of an action type.

The method uses the following notification category values in the built-in implementation.

- `cancel` – Cancellation notification configurations
- `nonrenew` – Non-renewal notification configurations
- `renewal` – Renewal notification configurations

The plugin must return the number of days lead time that represents the minimum value for the lead time column. This must be the minimum value for all combinations of line of business and jurisdiction for the category and effective date.

Notification by category (maximum)

There is another method with identical method signature as the previously-mentioned method except that the name is `getMaximumLeadTime`. This method must calculate the maximum lead time (not the minimum) for all combinations of line of business and jurisdiction.

```
int getMaximumLeadTime(Date effDate, Map<PolicyLinePattern, State[]> lineToJurisdictions,
                       NotificationCategory category) throws RemoteException;
```

Payment gateway configuration plugin

PolicyCenter supports electronic payment of the up-front payment for a policy job. The `PaymentGatewayConfigPlugin` payment gateway configuration plugin uses properties to provide configuration setting values for communication with an electronic payment service. The Payment Gateway plugin uses these values to communicate with an electronic payment service.

The base configuration implements the plugin in the Gosu `DefaultPaymentGatewayConfigPlugin` class. You can modify the plugin to set the configuration values that you require. Alternatively, implement your own instance of the plugin to customize PolicyCenter application logic. You can use script parameters to specify the configuration settings. Using script parameters avoids having to change the plugin code if your payment gateway changes. To change the value of a script parameter, in PolicyCenter, navigate to **Administration**→**Utilities**→**Script Parameters** and then click the parameter to edit.

The following Gosu code example demonstrates this approach.

```
class CustomPaymentGatewayConfig implements PaymentGatewayConfigPlugin {

    /*
     * The URL to the deployed PolicyCenter application
     * @return The value of the PolicyCenterURL script parameter
     */
    override property get PolicyCenterURL(): String {
        return ScriptParameters.PolicyCenterURL
    }
    ...
}
```

The `ScriptParameters.xml` file contains the definition of the `PolicyCenterURL` script parameter.

The following table lists the configuration properties and their purpose.

Property	Description
<code>PaymentGatewayConnectionTimeoutInSeconds</code>	The length of time to wait for a response from the payment gateway, in seconds.

Property	Description
PaymentGatewayGetURL	The URL for HTTP GET commands on the payment gateway. PolicyCenter redirects the user to this address to take payment with a new instrument when the <code>RedirectToPaymentGateway</code> property has a value of <code>true</code> .
PaymentGatewayPartner	The payment gateway partner.
PaymentGatewayPassword	The password to access the payment gateway.
PaymentGatewayPostConnectionPORT	The connection port for HTTP POST commands on the payment gateway, for example, 443.
PaymentGatewayPostURL	The URL for HTTP POST commands on the payment gateway.
PaymentGatewayTransactionType	The transaction type of the payment gateway command, as a <code>PaymentTransactionType</code> typecode. PolicyCenter supports only the <code>Sale</code> transaction type.
PaymentGatewayUser	The user to access the payment gateway.
PaymentGatewayVendor	The payment gateway vendor, also known as the merchant.
PolicyCenterURL	The URL that the payment gateway uses to return information to PolicyCenter, such as <code>http://mygpcserver:8180/pc</code> .
RedirectToPaymentGateway	Whether to redirect to the payment gateway for a payment that uses a new payment instrument. If this property is <code>false</code> , PolicyCenter informs the user that collection of electronic payment is not available.

See also

- “Implementing the payment gateway configuration plugin” on page 556
- Configuration Guide*

Payment gateway plugin

PolicyCenter supports electronic payment of the up-front payment for a policy job. To customize how PolicyCenter communicates with an electronic payment service, implement your own version of the payment gateway plugin interface `PaymentGatewayPlugin`. The plugin handles communication between PolicyCenter and any payment gateway that you need to support, such as PayPal or BarclayCard SmartPay. The interface methods provide the typical functionality of a payment gateway.

The base configuration implements the plugin in the Gosu `StandAlonePaymentGatewayPlugin` class. The plugin emulates the interaction with a payment gateway by running the PCF files in the `pcf/payment/demo` folder under the **Page Configuration** folder. You can examine this plugin to see the basic tasks that a payment gateway performs. Typically, you create your own implementation of this plugin to communicate between PolicyCenter and the payment gateways that you support.

The following table lists the required methods and their purpose.

Method	Description
<code>getCardDetails</code>	Gets a representation of the credit card as a payment instrument.
<code>inquiryPaymentGatewayTransaction</code>	Returns a payment gateway response object that contains information about the status of a transaction.
<code>isApprovedTransaction</code>	Returns a boolean value, which has a value of <code>true</code> if the specified transaction succeeded and <code>false</code> otherwise. This method uses the response object that <code>inquiryPaymentGatewayTransaction</code> returns to get this information.

Method	Description
<code>isVoidedTransaction</code>	Returns a boolean value, which has a value of true if the specified transaction has been voided and false otherwise. This method uses the response object that <code>inquiryPaymentGatewayTransaction</code> returns to get this information.
<code>mapSilentPostRequestToPaymentGatewayResponse</code>	Returns a payment gateway response object that maps the parameters from an incoming HTTP servlet request to payment gateway response properties. The servlet handles closure of the browser when the user is accessing the payment gateway.
<code>redirectToPaymentGateway</code>	Uses exit points to redirect from PolicyCenter to the supported payment gateways.
<code>refundFullAmountForTransaction</code>	Returns a payment gateway response object that contains information about an attempt to refund a credit card transaction. PolicyCenter can perform this action only before binding a job.
<code>requestSecureToken</code>	Returns a payment gateway response object that contains a secure token for communication with the payment gateway as part of the payment gateway response. Implement code for each payment gateway that you need to support.
<code>submitAccountVerification</code>	Returns a payment gateway response object that contains information about the status of a new credit card verification. If the payment gateway does not require this verification, create a successful response without calling the gateway.
<code>takePaymentUsingPaymentInstrument</code>	Returns a payment gateway response object that contains information about an attempt to use a payment instrument to make a payment.

See also

- “Implementing the payment gateway plugin” on page 556

Policy evaluation plugin

To customize how PolicyCenter raises underwriting issues (`UWIssue` object) on a policy period, implement the policy evaluation plugin interface, `IPolicyEvaluationPlugin`. PolicyCenter includes a built-in implementation that raises underwriting issues and removes orphaned underwriting issues. The built-in implementation accomplishes these tasks indirectly through other classes. You only need to modify or reimplement this plugin if you choose an alternate way of raising and removing underwriting issues. Most likely, you can use the built-in implementation of this plugin without any modifications. To raise and remove your underwriting issue types, you modify the evaluator classes that this plugin uses indirectly.

If however, you choose an alternate way of raising and removing underwriting issues, the remainder of this topic provides guidance.

The plugin interface contains only one method, called `evaluatePeriod`.

```
void evaluatePeriod(PolicyPeriod period, UWIssueCheckingPoint checkingPoint);
```

Before returning from this method, your method must add all relevant underwriting issues (`UWIssue` objects) on the period. Additionally, the method must remove all orphaned underwriting issues.

One of the parameters is an underwriting issue checking set typecode defined in the `UWIssueCheckingSet` typelist. The typecode describes a point at which PolicyCenter can evaluate a policy and generate an underwriting issue.

Each typecode contains a `Priority` property that indicates the order within the typelist. High priority checking set are evaluated first.

Use the typecode value to raise new issues appropriate for the policy. Next, remove any orphaned issues.

If you choose to modify this plugin, Guidewire recommends that your implementation use the `PolicyEvalContext` class to create and remove orphaned issues because these are complex and error-prone tasks. To add an issue, use the `addIssue` method on the `PolicyEvalContext` object.

The `addIssue` method finds an existing `UWIssue` with this type and key or, if no such issue exists, creates a new issue. If this method returns a pre-existing issue, this method marks it as touched by setting the `HumanTouched` property to `true`. Because pre-existing issues are marked touched, the `removeOrphanedIssues` method (see later in this topic) does not remove them.

For example, you can create a new issue with the following code.

```
var context = new PolicyEvalContext(period, checkingPoint)
var newIssue = context.addIssue( "PAHighValueAuto", "testissue", "this is a longer description", null )
```

Just as in the built-in implementation, you can use the `PolicyEvalContext` object to remove old issues using the `PolicyEvalContext.removeOrphanedIssues()` method. That method removes or marks as inactive all issues for which both of the following conditions are true.

- The issue existed at the time PolicyCenter created the context object.
- No previous call to the `addIssue` method on `PolicyEvalContext` affected this issue.

The method removes any issues that are open or marked no longer applicable. In contrast, for issues that are approved or declined, this method marks them as inactive.

See also

- *Configuration Guide*

Policy hold job evaluation plugin

PolicyCenter uses the `IPolicyHoldJobEvalPlugin` plugin to configure policy holds in the policy hold batch process. A batch process called Policy Hold Job Evaluation runs nightly to reevaluate the policy holds that changed and no longer apply to the job. You can also run this batch process manually. Refer to the **Batch Process Info** menu item in the **Server Tools** page.

By default, this batch process performs the following operations.

1. Checks if the policy hold was modified and no longer applies to the job.
2. If so, sends activity reminders to producers to retry quote or bind of the job.

To support this behavior, a policy hold (`PolicyHold`) contains an array of `PolicyHoldJob` entity instances. They keep track of jobs that are currently held by a policy hold and the last time the job evaluated against the policy hold.

A `PolicyHoldJob` consists of the following fields.

- `PolicyHold` – A foreign key to the policy hold
- `Job` – A foreign key to the job
- `LastEvalTime` – The last time this job was evaluated against this policy hold

A `PolicyHoldJob` is created when PolicyCenter evaluates the Policy Holds Type Filter checking set and determines that an underwriting issue needs to be raised. In the base configuration, PolicyCenter evaluates the checking set in the `DefaultUnderwriterEvaluator` class.

PolicyCenter updates `LastEvalTime` when the batch process reevaluates a job against a policy hold. The `PolicyHoldJob` row for that policy hold and job is deleted if the hold no longer applies to the job and an activity reminder is sent to the producer.

`IPolicyHoldJobEvalPlugin` exposes the methods that this batch process uses to find jobs that need to be reevaluated and how to evaluate it. This plugin interface has two methods.

To find jobs that need to be evaluated against the policy holds blocking them, implement the following method.

```
public IQueryBeanResult<PolicyHoldJob> findJobsToEvaluate();
```

In the base configuration, this plugin is implemented by `PolicyHoldJobEvalPlugin` and finds jobs that have any of the following qualities.

- Open jobs
- Jobs with policy periods with an active blocking policy hold
- Jobs not evaluated since the last time the policy hold changed

To evaluate a job against a policy hold, implement the following method.

```
public void evaluate(PolicyHoldJob policyHoldJob)
```

In the built-in plugin implementation, the `evaluate` method checks if the policy hold has been deleted or the hold changed to cause the job to no longer match. If so, the method creates an activity and assigns to the producer.

Policy number generator plugin

The `IPolicyNumGenPlugin` interface is responsible for generating a policy number for the current policy period. This number differs from the policy number associated with the root `Policy` object which is shared by all periods of the policy. The policy number for the current policy period is associated with a particular instance of a `PolicyPeriod` object and can vary from one period to another.

The plugin interface has a single method called `genNewPeriodPolicyNumber`.

```
genNewPeriodPolicyNumber(policyPeriod : PolicyPeriod) : String
```

The `policyPeriod` argument references the `PolicyPeriod` entity to generate a policy number for.

The method returns the generated policy number as a `String`.

The method is called as part of a `Submission` job when creating the first period of a policy and also when rewriting or renewing a policy. If the user attempts to rewrite a policy, the PolicyCenter user interface provides a choice between reusing the current policy number or generating a new one. Depending on the user's selection, the value of the `Rewrite.isChangePolicyNumber` is set appropriately. If set to `true`, PolicyCenter calls the plugin to generate a new number.

The base configuration implements the plugin in the `PolicyNumGenPlugin` class. This implementation is for demonstration purposes and must be configured prior to being placed in a production environment. Its single public method always generates a new and random policy number if the `policyPeriod` argument's `Status` field is set to the typecode value of `PolicyPeriodStatus.TC_LEGACYCONVERSION`. Otherwise, the `Job` field of the `policyPeriod` determines whether to generate a new policy number. If the `Job` is of type `Submission`, `Rewrite`, or `RewriteNewAccount`, a new and random policy number is generated. For all other types of `Job`, such as `Renewal`, the `policyPeriod` object's `PolicyNumber` field value is returned.

Policy numbers conform to field validator

PolicyCenter includes a feature called field validators that help you ensure accurate input within the web application user interface. In the case of policy numbers, field validators ensure that user input of a policy number exactly matches the correct format. The following example shows a policy number field validator.

```
<ValidatorDef name="PolicyNumber" value="[0-9]{3}-[0-9]{5}"  
description="Validator.PolicyNumber"  
input-mask="###-#####"/>
```

If you are implementing policy number generator code for the first time, you must configure the policy number field validator so that it matches the format of your policy number. After you first implement a policy number generator or later change the policy number format, coordinate changes to the field validators or user entry of policy numbers fails.

Policy payment plugin

To customize how PolicyCenter generates a list of available reporting plans for a policy, implement your own version of the policy payment plugin interface, `IPolicyPaymentPlugin`.

The plugin has only a single method, which is called `filterReportingPlans`. This method takes a `Policy` entity and an array of `PaymentPlanData` objects. Your method must return an array of payment plan data objects (`PaymentPlanData`), which collectively represents the subset of plans that are available for the policy. You can use properties on both the policy period and the payment plans to filter the payment plans to return. For example, you can use the `AllowsPremiumAudit` property on the policy period to determine whether to return payment plans that allow premium reporting.

You can return the same array as was passed in. You do not have to generate a new identical array if you do not need to perform any filtering.

In the base configuration, for a policy period that allows premium reporting, this method returns all payment plans. For a policy period that does not allow premium reporting, this method returns only non-premium-reporting payment plans.

The following example implementation filters payment plans. If the policy allows premium reporting, the method returns only premium-reporting payment plans. Otherwise, the method returns only installment plans.

```
class PolicyPaymentPlugin implements IPolicyPaymentPlugin {
    override function filterReportingPlans( policy : Policy, plans : PaymentPlanData[] ) : PaymentPlanData[] {
        var plansToReturn : List<PaymentPlanData> = {}
        for ( plan in plans ) {
            if ( period.AllowsPremiumAudit ) {
                if ( plan typeis ReportingPlanData ) {
                    plansToReturn.add( plan )
                }
            } else {
                if ( plan typeis InstallmentPlanData ) {
                    plansToReturn.add( plan )
                }
            }
        }
        return plansToReturn?.toTypedArray()
    }
}
```

If you need to examine the payment plan summary, the `PaymentPlanSummary` object has the following properties.

- `BillingId`
- `Name`
- `PaymentCode`
- `Installment`
- `Total`
- `Notes`
- `PolicyPeriod` – Foreign key reference to the policy period where the plan summary resides

Policy period plugin

The `IPolicyPeriodPlugin` policy period plugin handles a variety of tasks related to `PolicyPeriod` entities. PolicyCenter includes a built-in implementation of this plugin in the Gosu `PolicyPeriodPlugin` class. You can modify this plugin or implement your own instance of this plugin to customize PolicyCenter application logic.

Modifying plugin for a particular line of business

To modify the plugin behavior for a particular line of business, put line-specific functionality in the `PolicyLineMethods` implementation class. These classes have names with the LOB prefix with the pattern `LOBPolicyLineMethods`. For example, the `PolicyPeriodPlugin` policy period plugin implementation has a `postApplyChangesFromBranch` method. This method calls `postApplyChangesFromBranch` for each line in the policy period.

```
override function postApplyChangesFromBranch(policyPeriod : PolicyPeriod, sourcePeriod : PolicyPeriod) {
    for (line in policyPeriod.Lines) {
        line.postApplyChangesFromBranch(sourcePeriod)
```

```
    }
    ...
}
```

To implement functionality for the workers' compensation line of business, the `gw.lob.wc.WCPolicyLineMethods` class implements the `postApplyChangesFromBranch` method.

```
override function postApplyChangesFromBranch(sourcePeriod : PolicyPeriod) {
    ...
}
```

Setting written date for a transaction

If PolicyCenter needs to set the written date for a transaction, PolicyCenter calls the policy period plugin method `determineWrittenDate` to set the `Transaction.WrittenDate` property. This method takes a policy period and a transaction object. The policy period is the policy period that owns this transaction. The `Transaction.WrittenDate` property is the date for accounting purposes that the premium is considered as written.

The method returns a `java.util.Date` object that represents the written date.

In the built-in implementation of this plugin, the code checks whether the period state date is after the transaction's posted date. If so, the method returns the period start date. Otherwise, returns the transaction's posted date.

Getting the unassigned policy number display string

The policy period plugin defines a getter for the `UnassignedPolicyNumberIdentifier` property. This property returns a display string for the policy number if the policy number is not assigned. In the built-in implementation, this getter returns the value `Unassigned`. This is the value of the display key `PolicyPeriod.UnassignedPolicyNumberIdentifier`.

You can check the `PolicyPeriod.PolicyNumberAssigned` Boolean property to see if a policy number is assigned. In the base configuration, this value is `false` when the `PolicyPeriod.PolicyNumber` is `null`.

Specifying types to omit if copying a contractual period

Implement the `returnTypesToNotCopyOnNewPeriod` method to tell PolicyCenter which entity types must not be copied from one contractual period to the other during renewal, rewrite, and rewrite new account jobs.

For example, you have custom entities that you use for integration or accounting reasons. But you might not want to copy them from one period to another. For example, perhaps you might want to force recalculation of some values to get unique IDs in the new period.

The method takes no arguments and returns a set of entity types (`IEntityType`) to omit. You can define the set directly in-line using Gosu shorthand for set creation.

```
override function returnTypesToNotCopyOnNewPeriod(): Set<IEntityType> {
    return { RatingPeriodStartDate, Form, FormEdgeTable, FormAssociation }
}
```

Specifying types to omit if copying a branch

Implement the `returnTypesToNotCopyOnNewBranch` method to tell PolicyCenter which entity types must not be copied from a branch. PolicyCenter calls this method during renewal, rewrite, rewrite new account, audit, cancellation, issuance, policy change, and reinstatement jobs. PolicyCenter also calls this method during multi-version quoting and side-by-side quoting.

For example, you have custom entities that you use for integration or accounting reasons. But you might not want to copy them from one period to another. For example, perhaps you might want to force recalculation of some values to get unique IDs in the new period.

The method takes no arguments and returns a set of entity types (`IEntityType`). You can define the set directly in-line using Gosu shorthand for set creation.

After setting period window

Implement the `postSetPeriodWindow` method to customize behavior after a user changes the period window of a contractual period. PolicyCenter calls this method if a user changes the effective date or expiration date of a

contractual period. The method gets the `PolicyPeriod` of the new branch, the old start date, and the old end date. You can check the `PolicyPeriod.PeriodStart` and `PolicyPeriod.PeriodEnd` dates to get the current values if needed.

The built-in plugin implementation calls the `postSetPeriodWindow` method for each line of business in the policy period.

For example, the `WCPolicyLineMethods` class has a `postSetPeriodWindow` method. This method updates workers' compensation jurisdiction rating period start dates (RPSDs) after the period change.

This built-in behavior is probably sufficient for most situations and you do not need to modify it.

[After creating draft branch in new period](#)

Implement the `postCreateDraftBranchInNewPeriod` method to customize behavior after a user creates a draft branch in a new contractual period. PolicyCenter calls this for renewal jobs and rewrite jobs. The method takes a `PolicyPeriod` entity and returns nothing.

The built-in plugin implementation calls the `postCreateDraftBranchInNewPeriod` method for each line of business in the policy period.

For example, the `WCPolicyLineMethods` class has a `postCreateDraftBranchInNewPeriod` method. This method updates workers' compensation jurisdiction rating period start dates (RPSDs) after creating a draft in the new period.

[Before promoting a branch](#)

Implement the `prePromote` method to customize behavior before a branch is promoted. After a job completes and its draft branch becomes legally binding, PolicyCenter calls this method before promoting the draft branch. This method takes a `PolicyPeriod` entity and returns nothing.

[After handling a preemption](#)

Implement the `postCreateNewBranchForPreemption` method to customize behavior after a branch has been created to handle preemption. Preemption is name for the situation in which two concurrent jobs for a policy were based on the same branch. After the second job finishes, you must choose whether to apply changes from recently-bound jobs into the active job that is about to be bound. Otherwise, you must withdraw the current job.

In the user interface it might appear that any changes applied as part of preemption are merged into the active branch. However, PolicyCenter actually creates an entirely new branch with based-on revision set to the most recently bound `PolicyPeriod` in that contractual period. PolicyCenter applies changes from preempted branches (or multiple branches for multiple preemptions) to handle the preemption. After handling the preemption, PolicyCenter discards the active branch that the user was actively working on and was preempted by earlier bound changes.

This method is called after the new branch is created but before the old draft branch is discarded. The first method parameter is the new branch (a `PolicyPeriod`) created to handle the preemption. This new branch was cloned from the most recently bound branch and then PolicyCenter applies (merges) all changes from the preempted job. This new branch has a based-on revision property that is set correctly so there is no longer a preemption issue. The second argument is the draft branch that the user was actively working, which is the job that was preempted.

[After creating branch for new edit effective date in a policy change](#)

Implement the `postCreateNewBranchForChangeEditEffectiveDate` method to customize behavior after PolicyCenter creates a branch for a change to the edit effective date in a policy change.

To change the edit effective date during a policy change, PolicyCenter replaces the active branch that the user was actively working on with a new branch. PolicyCenter then discards the branch that the user was actively working on. PolicyCenter makes a new branch based on the most recently bound branch, copies the changes from the active branch that the user was working on into the new branch, and then discards the active branch that the user was working on.

This method is called after the new branch is created but before the old policy change branch is discarded. The first method parameter is the new branch (a `PolicyPeriod`) created for the change to the edit effective date. The second argument is the branch for the policy change before changing the edit effective date.

After applying changes from a branch

Implement the `postApplyChangesFromBranch` method to customize behavior after PolicyCenter applies changes from a branch. PolicyCenter calls this method in the following circumstances.

- When handling preemptions. PolicyCenter calls this method after applying changes from the preempted branch to the newly created branch.
- In a policy change when a user changes the edit effective date. PolicyCenter calls this method after applying changes from the policy change branch to the new branch with the changed edit effective date.

PolicyCenter calls this method in jobs that can be preempted. These jobs are primarily policy changes, but can also be audit, cancellation, reinstatement, and renewal jobs. Only policy change jobs can have their edit effective date changed.

The `postApplyChangesFromBranch` method takes two `PolicyPeriod` entity instances as arguments. The first argument, `policyPeriod`, is the period to apply changes to. The second argument, `sourcePeriod`, is the `PolicyPeriod` to generate changes from.

In the case of a preemption, the `sourcePeriod` is the original `PolicyPeriod` the user was working on in a job. For example, the user starts a policy change which adds a vehicle and two coverages to that vehicle; this is the `sourcePeriod` argument. This job is preempted by another job. The `policyPeriod` argument is the `PolicyPeriod` created for the preempting job. PolicyCenter applies the changes to `policyPeriod` by adding the vehicle and two coverages to the `policyPeriod` branch. Then PolicyCenter calls this plugin method.

Whether to apply property changes in slice or window mode

Implement the Boolean `canDateSliceOnPropertyChanged` method to determine whether to apply a property change to an object at a particular date, the slice date. This method returns `true` in most cases. In the base configuration, a few exceptions require window mode, such as exposures in general liability and workers' compensation. In slice mode, applying a property change on a date splits the object and causes it to automatically prorate its scalable values.

The built-in plugin implementation calls this method when changing the effective date in a policy change transaction. If this method returns `true`, changes that were introduced by the policy change are reapplied at the new effective date of the transaction.

For example, the `WCPolicyLineMethods` class has a `canDateSliceOnPropertyChanged` method. This method selectively returns `false` on certain objects, such as `WCCoveredEmployee` or `WCFedCoveredEmployee`.

Whether entity types appear in slice or window mode when applying differences

Implement the `returnTypesToNotDateSliceOnApplyDiff` method to determine which entity types do not have changes applied on a certain date, or window mode. In the base configuration, this method returns the few entity types that require window mode, such as jurisdictions in workers' compensation. For these entity types, changes are applied for the entirety of its existence, or not at all.

In the base configuration, this method is called when changing the effective date of a policy change.

The built-in plugin implementation calls the `TypesToNotDateSliceOnApplyDiff` method for each line of business in the policy period. For all lines, the `RatingPeriodStartDate` entity type is included in the results.

When copying a submission

Implement the `postCopyBranchIntoNewPolicy` to customize the behavior when copying a submission.

In the base configuration, this method is called after a new submission is created and the policy and policy period are cloned.

The built-in implementation cleans up the policy period and readies it for editing.

Policy plugin

You can customize how PolicyCenter determines whether the application can start various jobs, based on dynamic calculations on the policy. Define your own implementation of the `IPolicyPlugin` plugin to customize all jobs except submission. There is a built-in implementation of this plugin with default behaviors.

Can start cancellation

Implement the `canStartCancellation` method to customize whether cancellation can start, given a policy and the effective date of the cancellation. The method must return `null` if the policy can be canceled. If the policy cannot be canceled then the method returns a `String` containing an error message to display.

The built-in implementation of this plugin requires the following conditions to exist.

- There is a promoted `PolicyPeriod` that includes the effective date passed to the method.
- The user has permission to cancel the policy.
- The policy is not already canceled as of the effective date. This allows for canceling an already canceled policy on an earlier date.
- There is no open issuance job for the policy. This ensures that issuance and cancellation jobs are never both open at the same time.
- The policy does not have an open rewrite job.
- Current and future terms have not been archived.

Implement your own version of this plugin to customize this behavior.

Can start reinstatement

Implement the `canStartReinstatement` method to customize whether reinstatement can start, given a `PolicyPeriod` entity for the canceled period. It must return `null` if the policy can be reinstated, or if it cannot be reinstated then return a `String` describing an error message to display.

The built-in implementation of this plugin requires the following conditions to exist.

- This `PolicyPeriod` is the most recent bound branch for this contractual period.
- The policy is issued.
- The policy is canceled.
- The policy does not have a completed final audit.
- The policy does not have an open rewrite job, issuance job, or reinstatement job.
- The user has permissions to reinstate this period.
- The reinstatement period does not overlap any existing bound period.
- Current and future terms have not been archived.

Can start rewrite

Implement the `canStartRewrite` method to customize whether rewrite can start, given a `Policy` entity and the effective and expiration dates for the rewrite job. The method returns `null` if the policy can be rewritten. If the policy cannot be rewritten then the method returns a `String` describing an error message to display.

The built-in implementation of this plugin requires the following conditions to exist.

- The user has the permission to rewrite.
- There is a bound `PolicyPeriod` entity in the contractual period for the provided effective date.
- The policy period must be canceled.
- The rewrite effective date is equal to or after the cancellation date.
- There are no other jobs open on the policy except audits.
- The policy is issued.
- The new contractual period would not overlap with any bound existing one.
- The current term is not archived.

Can start policy change

Implement the `canStartPolicyChange` method to customize whether a policy change can start, given a `Policy` entity and the effective date for the policy change job. It must return `null` if the policy can change at that effective date, or if it cannot change then return a `String` describing an error message to display.

The built-in implementation of this plugin requires the following conditions to exist.

- The user has permission to change the policy.
- There is a bound `PolicyPeriod` entity in the contractual period for the provided effective date.
- The based on `PolicyPeriod` is in effect at the `effectiveDate`.
- The policy does not have an in-progress rewrite job or issuance job.
- Current and future terms have not been archived.
- The policy is issued.

By default, PolicyCenter does not prevent concurrent policy changes. PolicyCenter only warns the user. If you want to completely prevent concurrent policy changes in the same contractual period (rather than just warn the user), implement this behavior in the `canStartPolicyChange` method.

[Can start audit](#)

Implement the `canStartAudit` method to customize whether an audit job can start, given a `Policy` entity and the effective date for the audit job. The method returns `null` if the policy can change at that effective date. If the policy cannot change, then the method returns a `String` describing an error message to display.

The built-in implementation of this plugin requires the following conditions to exist.

- The policy is a monoline workers' compensation policy.
- The user has permission to create and complete an audit job.
- The policy is issued.
- The current term is not archived.

[Can start issuance](#)

Implement the `canStartIssuance` method to customize whether an issuance job can start, given a `Policy` entity. It must return `null` if the policy can issue at that effective date, or if it cannot issue then return a `String` describing an error message to display.

The built-in implementation of this plugin requires the following conditions to exist.

- The policy contains a bound `PolicyPeriod` entity, and finds the most recently bound contractual period if there is more than one.
- That policy is not issued.
- The user has permission to issue the policy.
- That policy is not canceled.
- The user has the permission to issue a policy.
- There are no other open jobs.
- The current term is not archived.

[Can start renewal](#)

Implement the `canStartRenewal` method to customize whether a renewal job can start, given a `Policy` entity. the method returns `null` if the policy can renew at that effective date. If the policy cannot renew, then the method returns a `String` describing an error message to display.

The built-in implementation of this plugin requires the following conditions to exist.

- The user has permission to create and complete a renewal job.
- The policy is issued.
- The policy is not already canceled.
- There is no open rewrite or renewal job.
- There is a bound branch for this policy.
- The current term is not archived.

Utility functions in the base configuration plugin

The base configuration of the plugin has utility methods that are not in the plugin definition. Use or copy these methods as necessary.

- `appendIfFalse` – Given a Boolean condition, appends an error message `String` to an existing `StringBuilder` object. This is useful to make concise and readable code that tests a condition.
- `errorString` – Given a `StringBuilder` instance, converts to a `String` and omits any final comma.

Policy term plugin

The `IPolicyTermPlugin` policy term plugin calculates values related to policy terms. PolicyCenter includes a built-in implementation of this plugin in the `Gosu PolicyTermPlugin` class. You can modify this plugin to customize it. Alternatively, implement your own instance of this plugin to customize PolicyCenter application logic.

Calculate the period end date from a term type

Implement the `calculatePeriodEnd` method which returns the default expiration date as a `java.util.Date` object.

```
override function calculatePeriodEnd( effDate : Date, term : TermType, policyPeriod : PolicyPeriod ):  
    Date
```

The input parameters to `calculatePeriodEnd` are described below.

- `effDate` – The effective or start date of the policy period
- `term` – The policy term type in the form of a `TermType` typecode. The value must be the `policyPeriod.TermType` unless the `policyPeriod.TermType` is `null`.
- `policyPeriod` – The policy period (`PolicyPeriod` object)

This plugin method must set the expiration date for every `TermType` value. In the built-in implementation, term type possible values are `Annual`, `HalfYear`, and `Other`. You can extend this typelist as desired.

Guidewire recommends you make the calculations in this method simple, such as adding a fixed number of years, months, or days. If you make the calculations too complex, you might need to check for edge case conditions in other PolicyCenter code. Consider explaining these conditions in the user interface so users understand how these values were calculated.

Date reconciliation in the built-in implementation

Certain term types, such as half-year terms, have varying numbers of days. Date reconciliation ensures that, in the start month of the initial policy period, a new policy period starts on the same day of the month as the initial policy period. For example, date reconciliation ensures that two half-year terms provide coverage for one year exactly. You may need to implement date reconciliation for your custom term types.

This topic describes how the `calculatePeriodEnd` method in the built-in plugin implementation (`PolicyTermPlugin`) does date reconciliation for half-year terms. In some calculations, the built-in implementation uses the initial policy period start date, the start date of the first policy period for this policy.

Assume a policy with a half-year term where the first policy period starts on day X of a month. The `calculatePeriodEnd` method provides date reconciliation for the policy period end date by finding the first matching condition.

Condition	Period end
1. If the initial policy period start date is day X of a month	The policy period end date is day X unless one of the following conditions is true.
2. If the initial policy period start date is day X of a month and the policy period end date is in a month with fewer than X days	The policy period end date is the last day of the month.
3. If the initial policy period start date is day X of a month, and X is the last day of the month	Subsequent policy period end dates are the last day of the month.

For example, the first half-year term of a policy starts on August 30, 2018. The month of August has 31 days. Date reconciliation ensures two half-year terms provide coverage for a full year.

Period start	Period end	Description
August 30, 2018	February 28, 2019	Condition 2 applies. The period end month, February, has fewer than 30 days, so the period ends on the last day of February.
February 28, 2019	August 30, 2019	Condition 1 applies. Conditions 2 and 3 are not relevant. August is the same month that the initial policy period started, so the period ends on the same day.

The built-in plugin uses the `policyPeriod` parameter for date reconciliation. The `calculatePeriodEnd` method uses the `shouldPerformDateReconciliation` flag on `PolicyPeriod.Job` to determine whether to reconcile the date. By default, the flag is `false`. However, the submission, renewal, rewrite, and rewrite new account jobs set the flag to `true`.

Calculate a term type from period start and end dates

Implement the `calculatePolicyTerm` method to determine the term type of a policy term given its start date, a period end date, and a Product. The method must return a `TermType` typecode.

To make date calculations time-insensitive, consider using the `PolicyPeriod` method `trimToMidnight` to trim excess time values backward to midnight.

Some lines of business allow extra days of coverage at the end of an annual policy term (`TermType` is `Annual`) just in case the policy has not yet been renewed. You can add additional days to an annual term by implementing the `AdditionalDaysInAnnualTerm` property in the `PolicyLineMethods` implementation class for one or more lines of business. These classes have names with the LOB prefix with the pattern `LOBPolicyLineMethods`. In the base configuration, an annual workers' compensation policy provides coverage for one year plus 16 days by defining this property in the `WCPolicyLineMethods` class.

```
override property get AdditionalDaysInAnnualTerm() : int {
    return 16
}
```

Calculate period end from a based on PolicyPeriod

Implement the `calculatePeriodEndFromBasedOn` method to return the default expiration date based on the effective date, the term type, and a based-on `PolicyPeriod`. This method must always return a non-null date. The return value is a `java.util.Date`.

The method accepts the following arguments.

- `effDate` – The effective or start date of the policy period
- `term` – The policy term type in the form of a `TermType` typecode. Set the expiration date for every `TermType` value and throw an exception for unexpected term types. In the built-in implementation, term type possible values are `Annual`, `HalfYear`, and `Other`. You can extend this typelist as desired.
- `policyPeriod` – A `PolicyPeriod` object that a new period is based on
- `shouldPerformDateReconciliation` – A Boolean value that specifies whether to perform date reconciliation. Date drifting occurs when a policy term of a given `TermType` (or multiple policy terms) does not cover the expected number of days. For example, two consecutive policy terms with the term type `HalfYear` would provide exactly one year of coverage, so date reconciliation ensures that behavior. The base configuration implements the following characteristics.
 - The end date of the second term lines up with the start date of the first term so that together, they cover one year just as an annual term would.
 - Consecutive annual terms naturally have the same start day, but are adjusted for leap year days.

For the returned date, the time part of the returned expiration date will be ignored in favor of the existing expiration time as set by the effective time plugin.

Proration plugin

PolicyCenter has the following uses for the proration plugin.

- Generating transaction calculations
- Rating integration
- Calling the `financialDaysBetween` method to calculate a date range that rating uses

PolicyCenter standardizes proration calculations in the proration plugin. For example, the transaction calculator uses the proration plugin rather than directly implementing its own proration.

The built-in implementation of the `ProrationPlugin` plugin implements the `IProrationPlugin` plugin interface. If you need to customize proration, you can create your own class that implements this plugin interface. For example, you can change how the plugin calculates the number of financial days between two dates.

Leap days and proration

In the base configuration, the proration plugin ignores leap days when calculating prorated premium if the `IgnoreLeapDayForEffDatedCalc` in `config.xml` is set to `true`. In the base configuration, this parameter applies to both prorated premiums and scalable fields.

If you need to calculate leap days differently for prorated premiums than for scalable fields, you can add an `IgnoreLeapDays` plugin parameter to the `IProrationPlugin`. The `ProrationPlugin` checks for this plugin parameter.

Add a plugin parameter to the proration plugin

Procedure

1. In Studio, navigate to **configuration**→**config**→**Plugins**→**registry**, and open the `IProrationPlugin.gwp` file.
2. Under **Parameters**, click  to add a parameter.
3. In the **Name** field, enter `IgnoreLeapDays`.
4. In the **Value** field, enter `true` or `false`.

Provide your own prorater subclass

Any implementation of the proration plugin interface must implement one method called `getProraterForRounding`. This method must return an instance of the class `gw.financials.Prorater`. The recommended way to implement this is for the plugin implementation to declare an inner class that extends the `Prorater` class.

The built-in implementation `gw.plugin.policyperiod.impl.ProrationPlugin` contains an inner class `ForGivenRoundingLevel`, which extends `Prorater`.

Your prorater subclass

Your `Prorater` implementation must override some methods of `Prorater` and can optionally override others, as described in the following table.

Prorater method	Required to override?	Description
<code>construct</code>	Required	Sets private variables to store the rounding level and rounding mode from constructor parameters. Your constructor must call <code>super</code> .
<code>prorateFromStart</code>	Required	Prorates an amount of money. Its parameters are listed below. <ul style="list-style-type: none"> • A period start date • A period end date • A date to prorate to

Prorater method	Required to override?	Description
		<ul style="list-style-type: none"> An amount as a <code>BigDecimal</code> <p>Returns the new amount as a <code>BigDecimal</code> value.</p>
<code>scaleAmount</code>	Optional	<p>Scales an amount with the rounding level and rounding mode passed to the constructor. There are two method signatures for this method.</p> <ul style="list-style-type: none"> One method signature takes a <code>BigDecimal</code> value and returns a <code>BigDecimal</code> value. One method signature takes a <code>MonetaryAmount</code> value and returns a <code>MonetaryAmount</code> value. <p>If you override <code>scaleAmount</code> method that takes a <code>MonetaryAmount</code>, you might also need to override the <code>BigDecimal</code> version. The default implementation of the <code>MonetaryAmount</code> version of the method calls the <code>BigDecimal</code> version of the method. In contrast, if you modify the <code>BigDecimal</code> version, you do not need to update the <code>MonetaryAmount</code> version of the method</p>
<code>toString</code>	Optional	Returns a description of this proration engine.
<code>financialDaysBetween</code>	Optional	Calculates the number of financial days between two dates. In the base configuration, includes or ignores a leap day based on configuration parameters.
<code>findEndOfRatedTerm</code>	Optional	Takes a start date and a number of days and returns the corresponding end date of the policy term. In the base configuration, includes or ignores a leap day based on configuration parameters.

The `Prorater` class has a method called `prorate` that PolicyCenter rating code calls to prorate an amount. The `prorate` method is public but is final, which means that you cannot override the method. The built-in implementation of this method calls the method `prorateFromStart` and implements standard proration.

In the base configuration, the `financialDaysBetween` and `findEndOfRatedTerm` methods ignore a leap day if either of the following conditions exists.

- The `IgnoreLeapDays` parameter on the plugin interface is `true`.
- The `IgnoreLeapDays` parameter is not specified, and the `config.xml` parameter `IgnoreLeapDayForEffDatedCalc` is `true`.

The `financialDaysBetween` and `findEndOfRatedTerm` methods are complementary. Therefore, a change in one method often necessitates a corresponding change in the other method. For example, in the following code, the values for `newDate` and `date2` must be the same after running the code.

```
var numDays = prorater.financialDaysBetween(date1, date2)
var newDate = prorater.findEndOfRatedTerm(date1, numDays)
```

How PolicyCenter rating code interacts with prorater subclasses

The PolicyCenter rating system generates `Cost` entity instances to represent costs of items on a policy. The `Cost.ActualAmount` property indicates the actual cost after proration. PolicyCenter calculates this cost by calling the `computeAmount` method on the corresponding `CostData` object for that type of cost. In the base configuration, the `computeAmount` method on a `CostData` class uses the prorater class that the proration plugin provides.

On a partial-term job such as `PolicyChange`, some rated amounts may change, so corresponding `Cost` entity instances must change, too. Typically, PolicyCenter adds partial-term offset and onset transactions.

- Offset transaction – Covers the old premium for the remainder of the term
- Onset transaction – Covers the new premium for the remainder of the term

If the `computeAmount` method of the `CostData` object and the prorater disagree about the `TermAmount` property and the effective date range, PolicyCenter must resolve the discrepancy. PolicyCenter assumes that the `Cost` entity instance is correct and the `Transaction` calculation is wrong. PolicyCenter generates full offset and onset transactions instead of partial ones, thus the offset and onset effective dates represent the entire period range, starting at the `PeriodStart` date.

See also

- For the relationship between `Cost` and `CostData`, see “Overview of cost data objects” on page 381

Quote purging plugin

PolicyCenter provides the `PurgePlugin` plugin interface so that you can modify the behavior of quote purging. The `Purge` batch process calls the methods and property getters in this plugin.

PolicyCenter provides a built-in implementation of this plugin, called `PCPurgePlugin`, which implements the `PurgePlugin` plugin interface. If you need to customize quote purging, you can create your own class that implements this plugin interface.

The base configuration of PolicyCenter uses a built-in implementation of the plugin, `PCPurgePlugin.gw`. To edit the registry for `PurgePlugin`, in Studio, navigate to `configuration`→`config`→`Plugins`→`registry`, and open the `PurgePlugin.gwp` file.

See also

- *Application Guide*
- *Configuration Guide*

Create context

The `createContext` method takes as input a `PurgeContext` object and returns a new `ExtendedContext` object. The `ExtendedContext` object is passed from the `prepareForPurge` method to the `postPurge` method. The `createContext` method sets up the object before it is passed to the `prepareForPurge` method.

In the base configuration, the `PCPurgePlugin` class simply returns the input `PurgeContext` object in the `createContext` method.

In your implementation of the `PurgePlugin` plugin interface, you can modify the `ExtendedContext` object. You can also create subclasses of this object.

In your implementation, you can also modify the `createContext` method. For example, based on business logic you can determine which subclass of the `ExtendedContext` object to return.

Prepare for purge

The `prepareForPurge` method takes as input the `PurgeContext` object. This method can modify the `PurgeContext` object and take actions before the batch process evaluates whether the job can be purged or pruned. The `PurgeContext` object is passed to the `postPurge` method.

In the base configuration, the `PCPurgePlugin` class stores the public ID of the object to be considered for purging in the `PurgeContext` object the `prepareForPurge` method.

In your implementation of the `PurgePlugin` plugin interface, you can modify the `prepareForPurge` method to take actions before the work queue evaluates whether to purge the policy period. For example, you might modify the `prepareForPurge` method to perform the following operations.

- Make changes to the policy period
- Write to the log file
- Gather statistics from the policy period

Guidewire does not support configuring quote purging to remove archived policy periods.

Take actions after purge

The `postPurge` method takes actions after the policy period has been purged. This method is called only if the job is purged. The `PurgeContext` is passed into this method.

In the base configuration, the `PCPurgePlugin` class takes no actions in the `postPurge` method.

In your implementation of the `PurgePlugin` plugin interface, you can modify this method to take actions after purge such writing to the log file a message that the purge completed.

Skip policy period for purge

The `skipPolicyPeriodForPurge` method takes as input a `PolicyPeriod` and returns a Boolean value indicating whether to skip purging the policy period. The method returns `true` to signal not purging a policy period.

In the base configuration, the `PCPurgePlugin` always returns `false` in the `skipPolicyPeriodForPurge` method.

In your implementation of the `PurgePlugin` plugin interface, you can modify this method to signal not purging certain types of policy periods.

Skip orphaned policy period for purge

The `skipOrphanedPolicyPeriodForPurge` method takes as input an orphaned `PolicyPeriod` and returns a Boolean value indicating whether to skip purging the policy period. Preempted jobs create orphaned policy periods, which are policy periods not associated with a job. To signal skipping an orphaned policy period, return `true`.

In the base configuration, the `PCPurgePlugin` class always returns `false` in the `skipOrphanedPolicyPeriodForPurge` method.

In your implementation of the `PurgePlugin` plugin interface, you can modify this method to signal skipping certain types of orphaned policy periods.

Get allowed job subtypes for purging

The `AllowedJobSubtypesForPurging` property getter returns a list of job subtypes that can be purged.

In the base configuration, the `PCPurgePlugin` class has the following job subtypes in the `AllowedJobSubtypesForPurging` property getter.

- `Submission`
- `PolicyChange`

In your implementation of the `PurgePlugin` plugin interface, you can add or remove job subtypes from the list.

Guidewire recommends that you not include the `Audit` job subtype. The final status of an audit job's `PolicyPeriod` is not `Promoted` or `Bound`. Therefore, the filtering criteria for the Purge batch process will not work as expected.

Get allowed job subtypes for pruning

The `AllowedJobSubtypesForPruning` property getter returns a list of job subtypes that can be pruned.

In the base configuration, the `PCPurgePlugin` class has the following the job subtypes in the `AllowedJobSubtypesForPruning` property getter.

- `Submission`
- `PolicyChange`
- `Renewal`

In your implementation of the `PurgePlugin` plugin interface, you can add or remove job subtypes from the list.

Guidewire recommends that you not include the `Audit` job subtype. The final status of an audit job's `PolicyPeriod` is not `Bound`. Therefore, the filtering criteria for the Purge batch process will not work as expected.

Calculate next purge check date

The `calculateNextPurgeCheckDate` method takes as input a job and calculates the date after which the job can be checked for purging by the Purge batch process. The `NextPurgeCheckDate` property is on the `Job`.

In the base configuration, the `PCPurgePlugin` class checks the `PurgeStatus` property of the `Job` and takes the following actions in the `calculateNextPurgeCheckDate` method.

PurgeStatus	Action
NoActionRequired	This status indicates that no purging is necessary. The code simply returns <code>null</code> .
Pruned	This status indicates that the job has no alternate versions to prune. This job is not in need of pruning, but the job may be in need of purging. Therefore the code calculates and returns the purge recheck date.

PurgeStatus	Action
Unknown	This status indicates that the job purge status is not known. This job may be in need of pruning or purging. Therefore the code calculates the prune and purge recheck dates. The code compares these two dates and returns the earlier date.

Get purge job date

The `getPurgeJobDate` method takes as input a `Job` and determines the date on or after which the job can be purged. The base configuration `PCPurgePlugin` plugin class performs the following operations in the `getPurgeJobDate` method.

- Calculates the purge date for closed jobs.
- Calculates the purge date based on the number of days that must pass after the end of a job's selected policy. This is defined in the `PurgeJobsPolicyTermDays` parameter.
If `PurgeJobsPolicyTermDaysCheckDisabled` is `true`, then this date is `null`.
- Compares the two purge dates, and return the later date. The job can be purged on or after this date.

Get prune job date

The `getPruneJobDate` method takes as input a `Job` and determines the date on or after which the job's unselected policy periods can be pruned.

In the base configuration, the `PCPurgePlugin` class performs the following operations in the `getPruneJobDate` method.

- Calculates the prune date for closed jobs.
- Calculates the prune date based on the number of days that must pass after the end of a job's selected policy. This is defined in the `PruneVersionsPolicyTermDays` parameter.
If `PruneVersionsPolicyTermDaysCheckDisabled` is `true`, then this date is `null`.
- Compares the two prune dates, and return the later date.

Disable purging archived policy periods

The `disablePurgingArchivedPolicyPeriods` method returns `true` because purging archived policy periods is disabled. In the base configuration, the `PCPurgePlugin` class always returns `true`.

In your implementation of the `PurgePlugin` plugin interface, always return `true` in the `disablePurgingArchivedPolicyPeriods` method.

Guidewire does not support configuring quote purging to remove archived policy periods.

Purge context object

The `gw.plugin.purge.PurgeContext` object is used to pass information before purging (`prepareForPurge` method) and after purging (`postPurge` method). You can extend this object by creating a subclass with additional properties and returning an instance of the subclass in place of `PurgeContext`. The base `PurgeContext` object and methods are not modifiable. The purge context specifies properties that identify the entity being purged and the entity's parents. These properties are listed below.

Property	Description
Policy	Contains one of the following items. <ul style="list-style-type: none"> • The policy being purged • The parent policy of the job being purged • The parent policy of the policy period being purged
Job	The job being purged or the parent job of the policy period being purged, if any.
PolicyPeriod	The policy period being purged.

Property	Description
PurgePolicy	Boolean. Value is true if a policy is being purged.
PurgeJob	Boolean. Value is true if a job is being purged.
PrunePeriod	Boolean. Value is true if a policy period is being purged.

You can extend the `PurgeContext` object by adding additional properties. For an example, see the `PurgePlugin` class which contains a static inner class `ExtendedContext`.

If you extend the `PurgeContext` object, Guidewire recommends that these properties contain data values rather than entity references. Never add a property with an entity reference belonging to the domain graph of the purged object because the purged object does not exist after the purge.

Reference date plugin

PolicyCenter calls the currently-registered `IReferenceDatePlugin` reference date plugin to evaluate the type of date to use to calculate reference dates for different types of objects.

For the default behavior, refer to the built-in implementation of this in the class `ReferenceDatePlugin`.

Each of the plugin methods returns a `Date` object.

Coverage reference date

In the `getCoverageReferenceDate` method, calculate and return the reference date for a coverage. The method takes a coverage pattern and a coverable.

Exclusion reference date

In the `getExclusionReferenceDate` method, calculate and return the reference date for an exclusion. The method takes a coverage pattern and a coverable. PolicyCenter uses this date for availability calculations.

Modifier reference date

In the `getModifierReferenceDate` method, calculate and return the reference date for a modifier. The method takes a modifier pattern and a modifiable object. PolicyCenter uses this date for availability calculations.

Period reference date

In the `getPeriodReferenceDate` method, calculate and return the reference date for a period. The method accepts the following arguments.

- A `PolicyPeriod` entity
- A reference date type, which is a `ReferenceDateType` typecode such as `EFFECTIVEDATE`, `RATINGPERIODDATE`, or `WRITTENDATE`
- A state of type `State`

You must return the date to use as the reference date for effective dating.

Renewal plugin

To customize how PolicyCenter handles renewals, implement `IPolicyRenewalPlugin`. For example, this plugin determines when to start a Renewal job for a `PolicyPeriod`. However, note that this plugin does not actually create the Renewal job itself. Instead, the work queue called `PolicyRenewalStart` actually starts the renewal job. A work queue is like a built-in batch process except that it automatically parallelizes the work across a PolicyCenter cluster. The renewal work queue looks for policy renewals to start.

The most important method in the `IPolicyRenewalPlugin` plugin interface is the `shouldStartRenewal` method. It determines whether to start the renewal job, as previously mentioned. Its only argument is a policy period and returns `true` if and only if the policy period is ready for a new renewal.

The built-in implementation evaluates the date that the renewal normally starts. To do this it calls another method on the plugin called `getRenewalStartDate`. If you write your own version of this plugin, refer to the built-in implementation and copy the code from the `getRenewalStartDate` method if appropriate.

If you write your own version of this plugin, refer to the built-in implementation class `gw.plugin.job.impl.PolicyRenewalPlugin`. Copy the `getRenewalStartDate` method as appropriate.

The plugin implemented in the base configuration performs the following operations.

- If there are open conflicting jobs on the policy, it prevents renewal until the user handles these issues.
- If there are no conflicting jobs, this method returns `true` if `getRenewalStartDate()` returns a date in the past.
- If there are conflicting jobs, this method returns `true` if `getRenewalStartDate()` returns a date less than three days in the past.
- All other cases return `false`.

Get renewal start date

The other method you must implement in this plugin is the `getRenewalStartDate` method. This method must determine the date on or after which to start a renewal job for a given `PolicyPeriod`. In contrast to the `shouldStartRenewal` method, the `getRenewalStartDate` method is primarily used for display purposes. It is also useful for testing purposes. Within this plugin, only the `shouldStartRenewal` method determines authoritatively whether the policy is ready for renewal.

The built-in implementation evaluates the given `PolicyPeriod` to determine the earliest date that PolicyCenter creates a renewal job for this policy period. This implementation gets a notification date for the `PolicyPeriod` from the currently-registered implementation of the `INotificationPlugin`. The built-in implementation then adds further lead time based on the product and period end date of the policy period. It always truncates the date to midnight on that date. In other words it removes the time-of-day part of that date. This happens because it forces any programmatic date comparison of the date that this plugin returns to always be a day (not time) comparison.

Note:

Changing the implementation of `getRenewalStartDate` does not reset the field `PolicyTerm.NextRenewalCheckDate` of existing policy terms. Therefore, run the Clear Policy Renewal Check Dates batch process if you modify the implementation of this method.

Get automated renewal user

Your plugin must be able to get the PolicyCenter user to associate the renewal policy period. Implement the `getAutomatedRenewalUser` method, which takes a `PolicyPeriod` and returns a `User`.

The built-in behavior finds and returns the PolicyCenter user with the credential `renewal_daemon`.

Determine whether to use renewal offers

Your plugin must be able to determine whether this is the kind of policy period appropriate for the PolicyCenter renewal offer flow. The built-in behavior checks if the current job type for this policy period is `Renewal` and also is a business auto product. You can customize this logic for each product to use all one flow, or a mix of flows depending on the product or other custom logic.

Underwriting company plugin

To customize how PolicyCenter finds which underwriting companies are available for a set of jurisdictions, implement your own version of the underwriting company plugin interface, `IUWCompanyPlugin`.

There is only one method on the plugin, which is `findUWCompaniesForStates`. Your implementation of this method must find all underwriting companies that are available for the set of jurisdictions (states) associated with this policy. You return the list of under writing companies as a set (`java.util.Set`) of underwriting companies (`UWCompany`).

Using Gosu generics notation, the type you need to return is `Set<UWCompany>`. If you want to filter the jurisdictions by product and effective date, you can choose to filter using those properties.

The method parameters are a policy period (`PolicyPeriod`) and a Boolean parameter called `allStates`. The usage of the `allStates` parameter is described below.

- If `allStates` is `true`, only return a company if the underwriting company contains all jurisdictions associated with the policy.
- If `allStates` is `false`, only return a company if the underwriting company contains any jurisdictions associated with the policy.

The built-in implementation of this plugin calls a built-in class to find underwriting companies by jurisdictions and product and valid on a specific date. It uses the `allStates` parameter, querying the `UWCompany` table with a conjunctive or disjunctive reverse join through the `LicensedState` table per state, depending on the value of the `allStates` value.

The following Gosu code example demonstrates this approach.

```
class UWCompanyPlugin implements IUWCompanyPlugin {  
    /**  
     * Finds all the underwriting companies that are available for the states in the set.  
     * Also allows filtering of the States by product and effective date.  
     * Does not return UWCompanies of "retired" status.  
     *  
     * @param period The Policy Period  
     * @param allStates If true, *all* states must be found on the UWCompany;  
     *                  if false, *any* must be found.  
     * @return A query of underwriting companies  
     */  
    override function findUWCompaniesForStates(period : PolicyPeriod, allStates : boolean)  
        : Set<UWCompany> {  
        // This OOTB implementation goes through Guidewire's UWCompanyFinder to retrieve the UWCompanies  
        // The finder queries on the UWCompany table with a conjunctive or disjunctive reverse join through  
        // the LicensedState table per state, depending on the value of allStates.  
        return PCDependencies.getUWCompanyFinder().findUWCompaniesByStatesAndProductAndValidOnDate(  
            period.AllCoveredStates, allStates, period.Policy.Product, period.PeriodStart).toSet()  
    }  
}
```


Authentication integration

To authenticate PolicyCenter users, PolicyCenter by default uses the user names and passwords stored in the PolicyCenter database. Integration developers can optionally authenticate users against a central directory such as a corporate LDAP directory. Alternatively, use single sign-on systems to avoid repeated requests for passwords if PolicyCenter is part of a larger collection of web-based applications. Using an external directory requires a user authentication plugin.

To authenticate database connections, you might want PolicyCenter to connect to an enterprise database but need flexible database authentication. Or you might 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 however necessary for your company.

Overview of user authentication interfaces

There are several mechanisms to log in to PolicyCenter.

- Log in using the web application user interface.
- Authenticate WS-I web service calls to the current server.

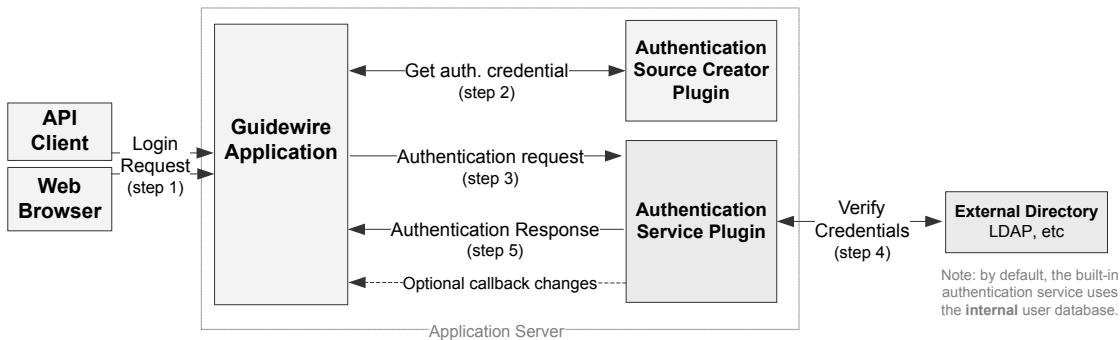
Authentication plugins must handle all types of logins other than WS-I web services.

The authentication of PolicyCenter 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 PolicyCenter, 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 PolicyCenter. Apache Tomcat can pass authentication-related properties to PolicyCenter 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 PolicyCenter.

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, PolicyCenter 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. PolicyCenter 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 PolicyCenter. PolicyCenter 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 – PolicyCenter 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 PolicyCenter 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, PolicyCenter sets up the user session and give the user access to the system. If rejected, PolicyCenter 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 PolicyCenter 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 PolicyCenter.

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 PolicyCenter 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 187

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. PolicyCenter 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 PolicyCenter, 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/pc/PolicyCenter.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 `java-api` directory after running the following command.

```
gwb genJavaApi
```

Extract the files in the archive file. The source files for the example class will be located in the following `PolicyCenter` directory.

```
java-api/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 knows how to 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 PolicyCenter. The JAAS example calls a JAAS provider to check credentials. Then it looks up the user's public ID in PolicyCenter 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 PolicyCenter.
- `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 PolicyCenter database. If the code finds a match in the database, the user is considered to be authenticated.
- The `JAAASAuthenticationServicePlugin` 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 `java-api` directory after running the following command.

```
gwb genJavaApi
```

Extract the files in the archive file. The source files for the example classes will be located in the following PolicyCenter directory.

```
java-api/examples/src/examples/pl/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 PolicyCenter 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 187

SOAP API user permissions and special-casing users

Guidewire recommends creating a separate PolicyCenter 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 PolicyCenter user interface.

From an authentication service plugin perspective, for those users you could create an exception list in your authentication plugin to implement PolicyCenter 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 PolicyCenter 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 `PolicyCenter/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 PolicyCenter generated libraries, place the libraries in `AUTHSOURCEROOT/lib/`.

Similarly, place your `AuthenticationService` plugin in the appropriate subdirectory of `PolicyCenter/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/basic/custom/authservice/MyAuthService.class`. If your code depends on any Java libraries other than PolicyCenter generated libraries, place the libraries in `AUTHSERVICEROOT/lib/`.

For PolicyCenter 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, PolicyCenter 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 PolicyCenter application. Users who connect to PolicyCenter using the API, for example, do 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 PolicyCenter 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 PolicyCenter, database authentication plugins help the PolicyCenter 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 PolicyCenter 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. PolicyCenter 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 PolicyCenter 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, PolicyCenter 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.

PolicyCenter also supports looking up database passwords in a password file by setting "passwordfile" as a `<database>` attribute in your main `config.xml` file.

At run time, the username and password returned by your database authentication plugin replaces the `${username}` and `${password}` parts of your database initialization `String` values.

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, PolicyCenter publishes the built-in `Login` web service.

Document management

PolicyCenter provides a user interface and integration APIs for creating documents, downloading documents, and producing automated form letters. You can integrate the PolicyCenter 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.

In PolicyCenter, the production of policy forms for printing services is unrelated to document management as described in this topic. There is a separate PolicyCenter forms API that is specific to the PolicyCenter product model.

The PolicyCenter user interface provides a **Documents** screen when you have a policy open.

The **Documents** screen lists documents such as letters, faxes, emails, and other attachments stored in a document management system (DMS).

Document storage overview

PolicyCenter 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 PolicyCenter 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 PolicyCenter web user. To address these issues, PolicyCenter 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 PolicyCenter: document content and document metadata. Each data type can be stored in different locations.

Type of data	Plugin interface and its default behavior
<p>Document content</p> <p>Document content can be anything that represents incoming or outgoing documents. For example:</p> <ul style="list-style-type: none"> • a fax image • a Microsoft Word file • a photograph • a PDF file <p>A document content source is code that stores and retrieves the document content.</p>	<p>IDocumentContentSource</p> <p>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.</p> <p>In a production environment, implement the IDocumentContentSource plugin to interact with a remote document management system.</p>
<p>Document metadata</p> <p>Document metadata describes the file and includes:</p> <ul style="list-style-type: none"> • a name, which typically is the file name • a MIME type • a description • the full set of metadata on the DMS so users can search for documents • the business objects that are associated with each document • other fields defined by the application or your extensions, such as recipient, status, or security type <p>A document metadata source is code that manages and searches document metadata.</p>	<p>IDocumentMetadataSource</p> <p>The base configuration implements an example plugin in the LocalDocumentMetadataSource 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.</p> <p>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.</p> <p>Alternatively, implement the IDocumentMetadataSource plugin to interact with a remote document management system.</p> <p>Metadata stored locally by PolicyCenter cannot later be stored by a remote DMS. Similarly, DMS-stored metadata cannot be moved to local PolicyCenter 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 PolicyCenter or a remote DMS. Changing the metadata storage mechanism invalidates the original hyperlink.</p>

Document storage plugin architecture

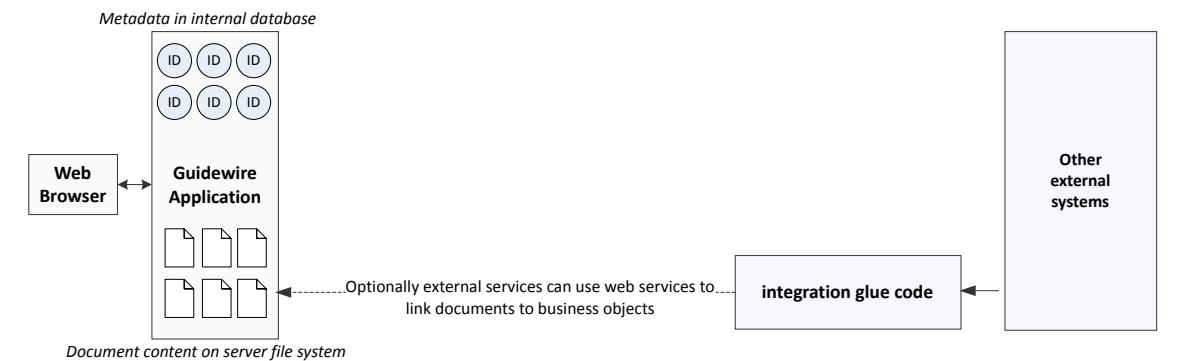
The following diagram summarizes implementation architecture options for document management.

Document Storage Architecture Options

KEY  Document content  Document metadata, such as file name, MIME type, and location of content

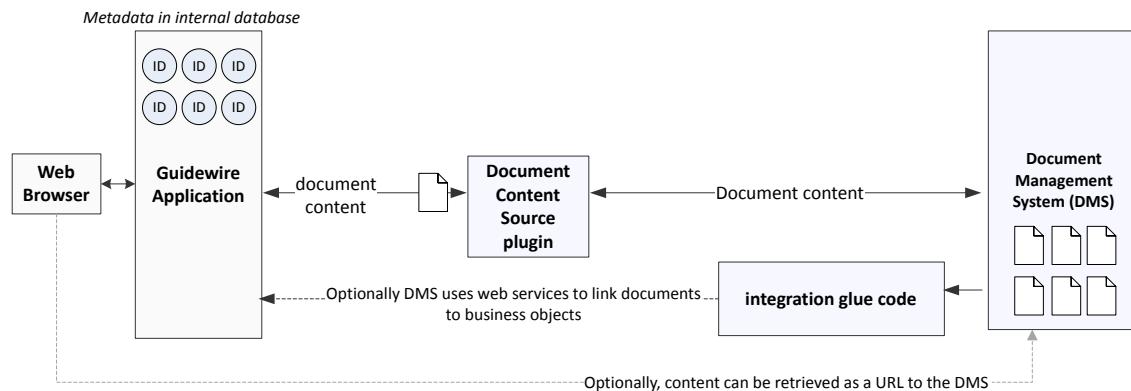
Internal Document Storage and Internal Metadata

Use built-in document content source plugin implementations. Disable the document metadata source plugin.



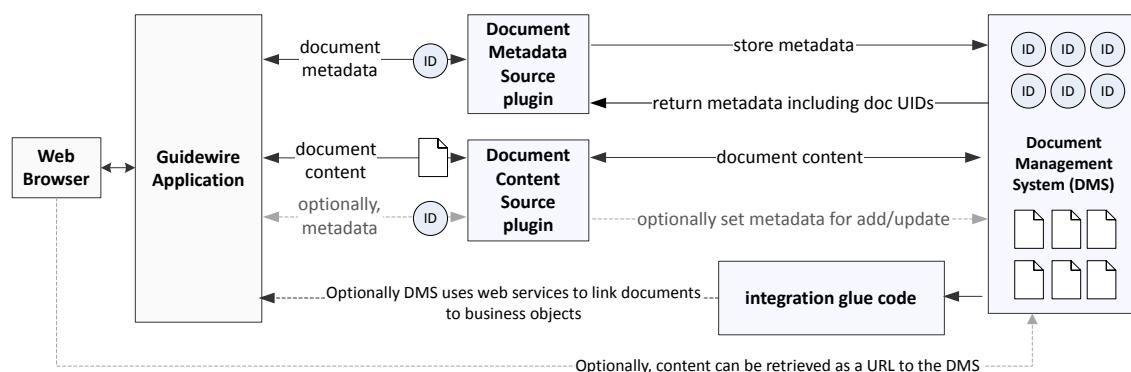
External Document Storage and Internal Metadata (Highest Performance Storage and Searching)

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 `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 PolicyCenter 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

PolicyCenter 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 PolicyCenter. 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, PolicyCenter 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 199
- “Add documents and metadata” on page 200
- “Retrieve documents” on page 201
- “Update documents and metadata” on page 203
- “Remove documents” on page 203
- “User interface elements when document management system unavailable” on page 203

Check for document existence

A document content source plugin must fulfill requests to check for the existence of a document. To check document existence, PolicyCenter 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.

PolicyCenter 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: PolicyCenter 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 PolicyCenter 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, PolicyCenter 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. However, PolicyCenter 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.
- The method can optionally set the modified date in the `Document.DateModified` property.

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`. PolicyCenter 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, PolicyCenter 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 PolicyCenter 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 PolicyCenter 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 PolicyCenter 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 PolicyCenter 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. PolicyCenter 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 204

Update documents and metadata

A new document content source plugin must fulfill requests from PolicyCenter 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. PolicyCenter 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`. PolicyCenter 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, PolicyCenter 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.

PolicyCenter 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 PolicyCenter. 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 205
- “Linking methods for the `IDocumentMetadataSource` plugin” on page 207

Basic methods for the `IDocumentMetadataSource` plugin

This topic covers the following methods:

- “`saveDocument`” on page 205
- “`retrieveDocument`” on page 205
- “`searchDocuments`” on page 206
- “`removeDocument`” on page 206
- “`isOutboundAvailable`” on page 206
- “`isInboundAvailable`” on page 207

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, PolicyCenter 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, PolicyCenter 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 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 207
- “`getDocumentsLinkedToEntity`” on page 207
- “`isDocumentLinkedToEntity`” on page 207
- “`unlinkDocumentFromEntity`” on page 207
- “`getLinkedEntities`” on page 208

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 PolicyCenter, the class `gw.plugin.document.impl.AsyncDocumentContentSource` implements the `IDocumentContentSource` plugin. This plugin implementation stores documents on the PolicyCenter server's file system. By default, PolicyCenter 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 208
- “Remember to store public IDs in the external system” on page 209

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 PolicyCenter 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 PolicyCenter system.

Documents are partitioned into subdirectories by policy and by relationships within the policy. The following table explains this directory structure.

Entity	Directory naming rules	Example
Account	Account + accountPublicID	/documents/AccountABC:02154/
Policy	accountDirectory + "/" + Policy + policyPublicID	/documents/AccountABC:02154/PolicyABC:02154/
PolicyPeriod	policyDirectory + "/" + PolicyPeriod + policyPeriodPublicID	/documents/AccountABC:02154/PolicyABC:02154/PolicyPeriodABC:02154/

Note: For all entities, the public ID is used. No other types of ID are used.

PolicyCenter 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, PolicyCenter 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 PolicyCenter. 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 a policy document:

/documents/policyABC:02154/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.

PolicyCenter 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 PolicyCenter 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 PolicyCenter user.

To address these issues, PolicyCenter 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.

PolicyCenter 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 PolicyCenter 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 PolicyCenter 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.

Config- ura- tion name	Description	Configuration	Com- pa- tible 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>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 true. 	No

Config- ura- tion name	Description	Configuration	Com- pa- tible with IDMS
	<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 PolicyCenter and ContactManager.</p>		
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 PolicyCenter can enforce limitations on final documents.

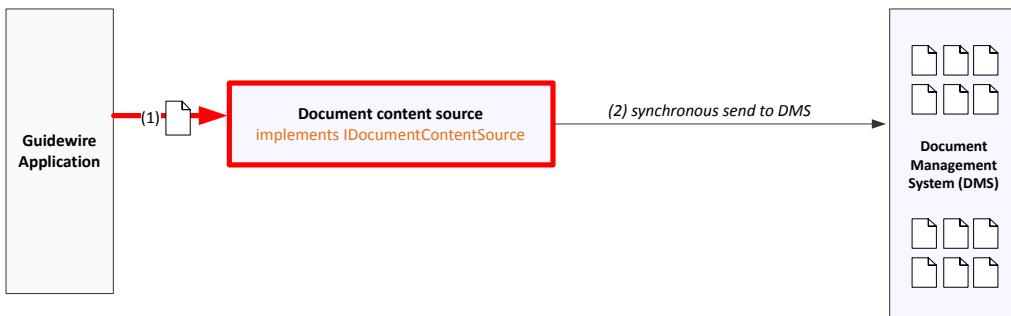
The following diagram contrasts the three options for document storage asynchronous configuration.

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.

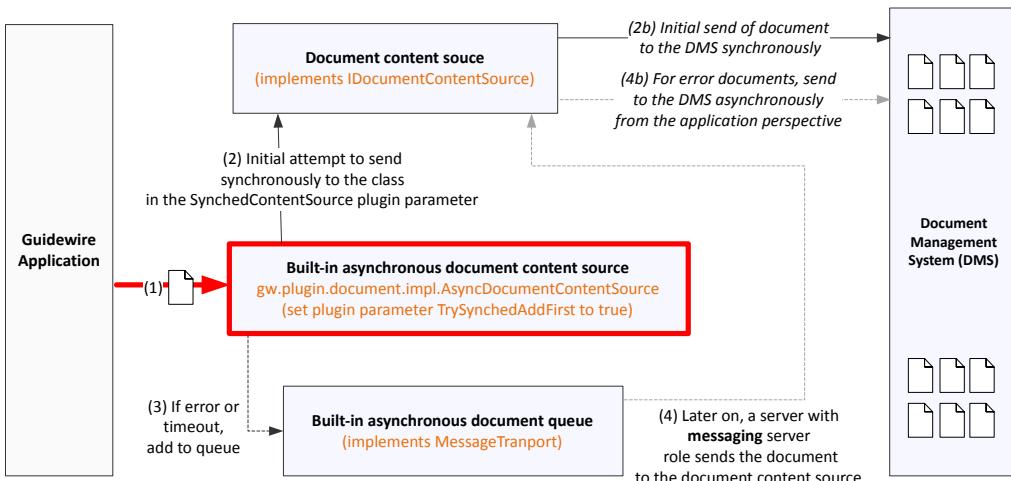
"Sync-only"

Direct synchronous connection to the `IDocumentContentSource` plugin implementation that connects to the DMS



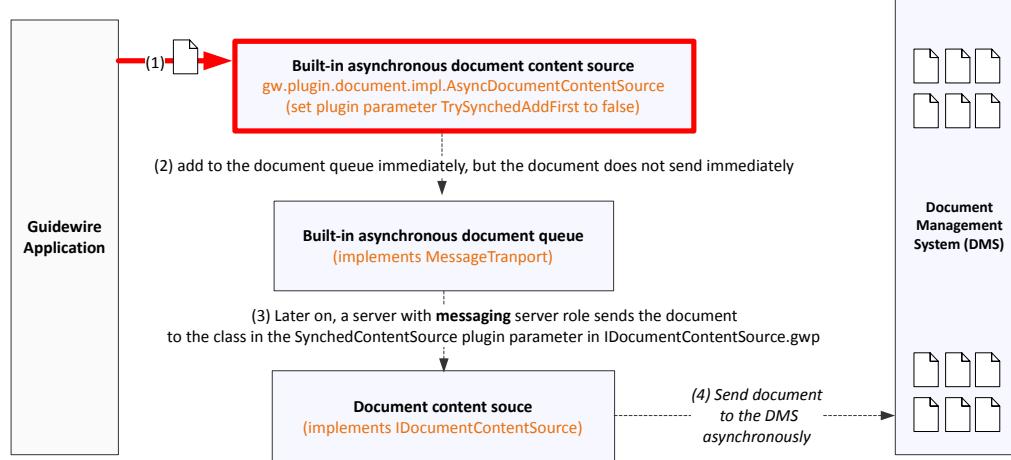
"Sync-first"

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 PolicyCenter 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 PolicyCenter 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, PolicyCenter 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, PolicyCenter 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 (PolicyCenter 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

PolicyCenter 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 PolicyCenter receives a document from the DMS, PolicyCenter 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, PolicyCenter 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 PolicyCenter user interface prevents metadata changes as well as uploading a new version of the document to the DMS.
- 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, PolicyCenter 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 PolicyCenter, 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

PolicyCenter 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 PolicyCenter, you can attach a new `Document` object to a `PolicyPeriod` object. On the `Document` entity instance, set the `Policy` property to the policy and the `PolicyPeriod` property to the policy period.

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

PolicyCenter provides a web services API for linking business objects (such as a policy) to a document. This allows external process and document management systems to work together to inform PolicyCenter 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 PolicyCenter of the new document and create a new PolicyCenter activity to review the new document. Similarly, after sending outbound correspondence such as an email or fax, PolicyCenter 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, PolicyCenter stores that document ID.

The `PolicyPeriodAPI` web service includes a method to add documents.

- `addDocumentToPolicyPeriod` – Add a new document to an existing policy.

See also

- “Policy period web services” on page 124

Document production

PolicyCenter provides a user interface and APIs to create documents, download documents, and produce automated form letters. This topic discusses creating new documents from templates.

In PolicyCenter, the production of policy *forms* for printing services are unrelated to document management as described in this topic. There is a separate PolicyCenter forms APIs that is specific to the PolicyCenter product model.

PolicyCenter 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. PolicyCenter can create some types of new documents from a server-stored template without user intervention.

In contrast to PolicyCenter, 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 PolicyCenter, users can create new documents from a template and then attach them to a policy or other PolicyCenter object. This is useful for generating forms or form letters or any type of structured data in any file format. Or, in some cases PolicyCenter creates documents from a server-stored template without user intervention. For example, PolicyCenter creates a PDF document of an outgoing notification email and attaches it to the policy.

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, PolicyCenter 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. PolicyCenter 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 <code>FormField</code> 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.

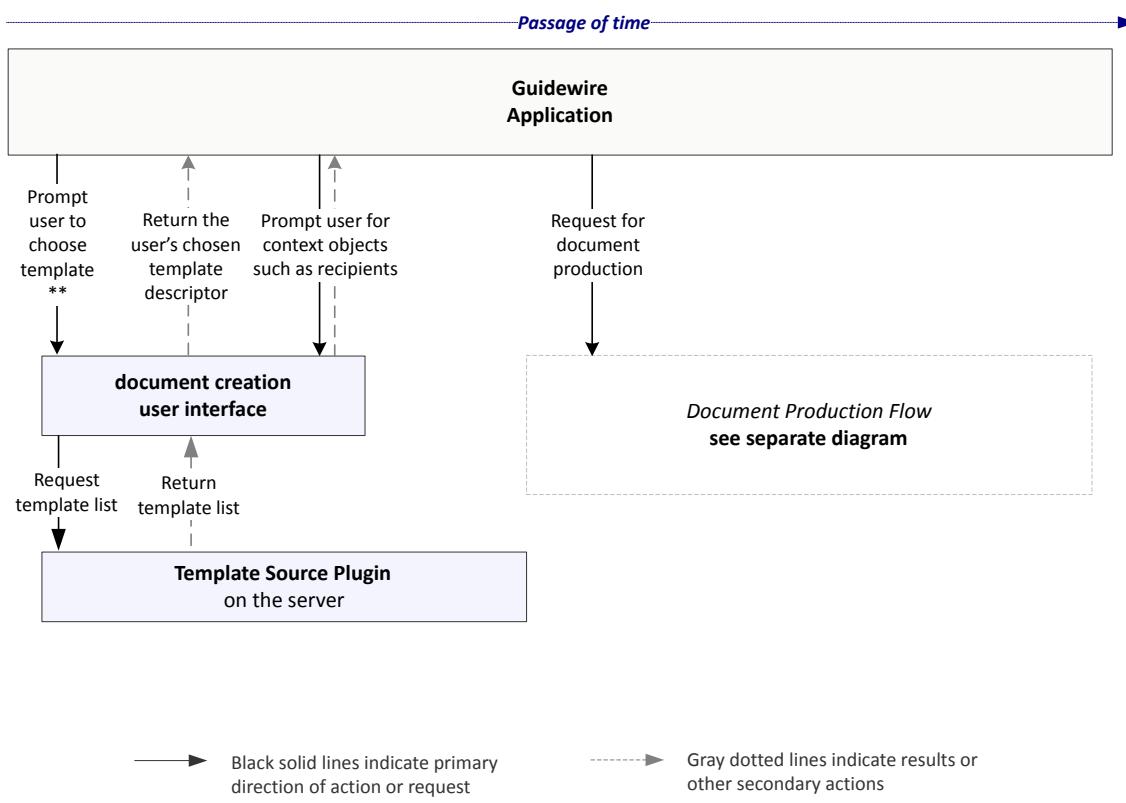
User interface flow for document production

From the PolicyCenter 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. PolicyCenter supports Gosu-initiated automatic document creation from business rules. The automatic and manual processes are similar, but the automatic

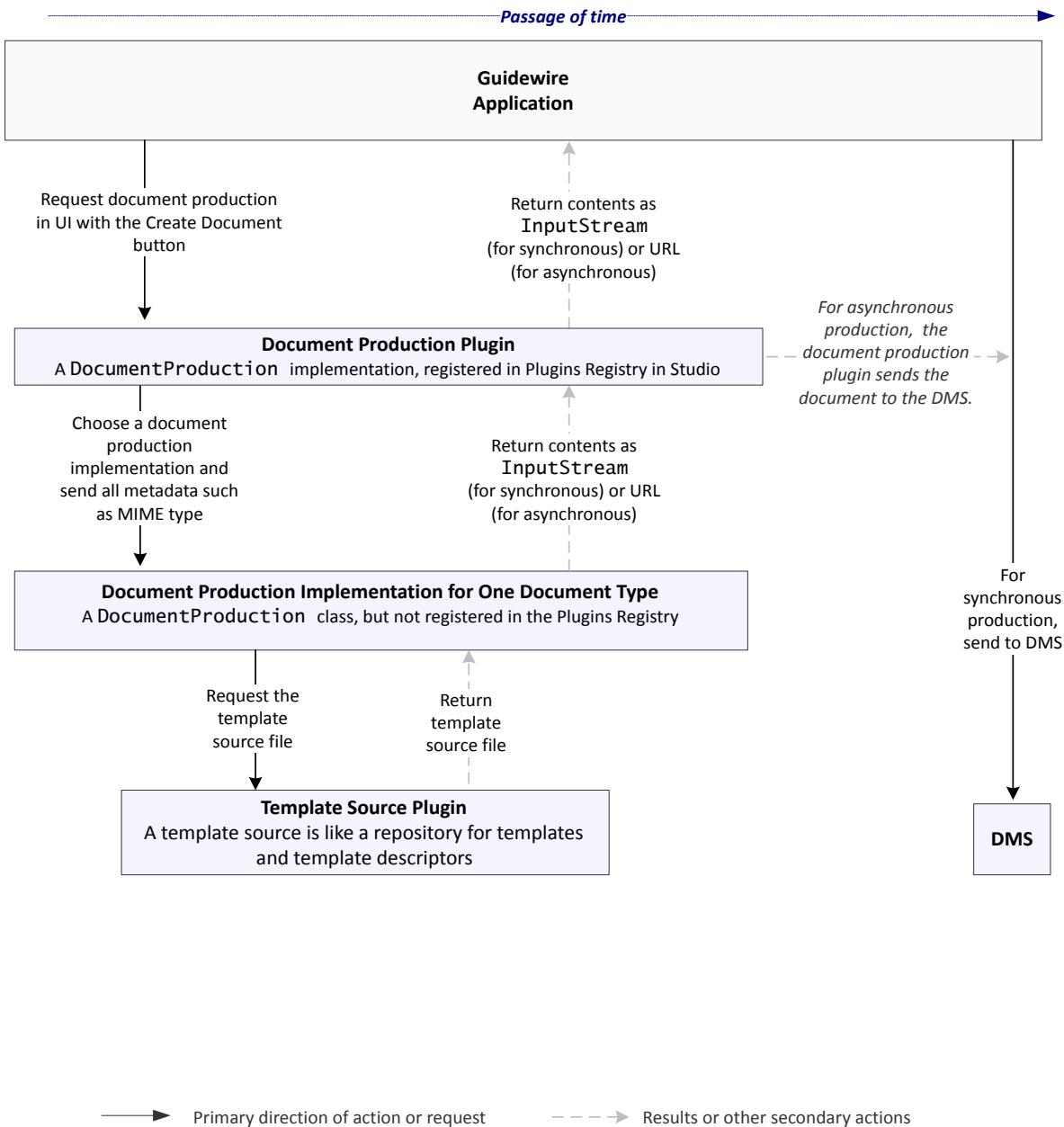
document creation skips the first few steps of the process. The automated creation process skips some steps relating to parameters that the user chooses in the user interface. Instead, the Gosu code that requests the new document sets these values.

The following diagrams illustrate interactions between PolicyCenter, the various plugin implementations, and an external document management system during document generation.

Interactive Document Creation UI Flow



Document Production Flow



The chronological steps are described below.

1. PolicyCenter invokes the document creation user interface to let you select a form or letter template.
2. The PolicyCenter 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 PolicyCenter.
6. PolicyCenter 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 PolicyCenter.
7. PolicyCenter 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. PolicyCenter 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 PolicyCenter business objects with PolicyCenter 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.

PolicyCenter 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, PolicyCenter 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, PolicyCenter provides interfaces that define the core functionality. In the base configuration, PolicyCenter 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 198

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 PolicyCenter that the document now exists.

If your code to add the document is running on the PolicyCenter server, use methods on the entity to add the document, as shown in the following code statement.

```
newDocumentEntity = Policy.addDocument
```

In either case, immediately throw an exception if any part of your creation process fails.

See also

- “APIs to attach documents to business objects” on page 216

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, PolicyCenter 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. PolicyCenter 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, PolicyCenter proceeds with document production using the specified `IDocumentProduction` implementation class in that plugin parameter value.
2. If PolicyCenter does not find a match for the document production type, PolicyCenter searches the plugin parameters for a MIME type that matches the `MimeType` property of the template. If a match is found, PolicyCenter 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. PolicyCenter uses this information to map between MIME types and file extensions. To map from a MIME type to a file extension, PolicyCenter uses the first extension in the list. To map from file extension to MIME type, PolicyCenter 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/pc/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 224
- “Licensing for PDF document production” on page 225

Licensing for Microsoft Excel and Word document production

For server-side document production of Microsoft Word and Microsoft Excel files, PolicyCenter includes license files in the following paths.

```
PolicyCenter/configuration/config/security/excel.lic  
PolicyCenter/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 PolicyCenter 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 or sending an email to support@guidewire.com and request a PDF production license for PolicyCenter.
2. Fill out the Authorization Form.
3. Fax 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, email support@guidewire.com.
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 `Policy` symbol is required, only application contexts that provide a `Policy` 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 policy.
- 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 PolicyCenter 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	Re- quired?	Description
<code>id</code>	Required	<p>The unique ID of the template. In the <code>IDocumentTemplateDescriptor</code> interface, this is the <code>templateId</code> property.</p>
<code>name</code>	Required	A human-readable name for the template.
<code>identifier</code>	Optional	<p>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.</p>
<code>scope</code>	Optional	<p>The contexts that this template supports. The following values are supported.</p> <ul style="list-style-type: none"> • <code>ui</code> – The document template must only be used from the document creation user interface. • <code>gosu</code> – The document template must only be used from rules or other Gosu and must not appear in a list the user interface template. • <code>all</code> – The document template may be used from any context.
<code>description</code>	Required	A human-readable description of the template and/or the document it creates.
<code>mime-type</code>	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>. In the <code>IDocumentTemplateDescriptor</code> interface, this is the <code> mimeType</code> property.</p>
<code>production-type</code>	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>. 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. In the <code>IDocumentTemplateDescriptor</code> interface, this is the <code>documentProductionType</code> property.</p>
<code>date-modified</code>	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. In the <code>IDocumentTemplateDescriptor</code> interface, this is the <code>dateModified</code> property.</p>
<code>date-effective date-expires</code>	Required	<p>The effective and expiration dates for the template. If you search for a template, PolicyCenter 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. In the <code>IDocumentTemplateDescriptor</code> interface, these are the <code>dateEffective</code> and <code>dateExpiration</code> properties.</p>
<code>keywords</code>	Required	A set of keywords to search for within the template. Delimit keywords with the comma character. Do not add space characters between keywords.

Document template descriptor attribute as it appears in the descriptor XML file that you must create for new templates	Required?	Description
required-permission	Optional	The code of the SystemPermissionType 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). In the <code>IDocumentTemplateDescriptor</code> interface, this is the <code>requiredPermission</code> property.
mail-merge-type	Optional	Optional configuration of pagination of Microsoft Word production. By default, PolicyCenter 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. In the <code>IDocumentTemplateDescriptor</code> interface, this is the <code>mailmergetype</code> property.
type	Required	Corresponds to the <code>DocumentType</code> typelist. Documents created from this template have their <code>type</code> fields set to this value. In the <code>IDocumentTemplateDescriptor</code> interface, this is the <code>templateType</code> property.
default-security-type	Optional	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> . In the <code>IDocumentTemplateDescriptor</code> interface, this is the <code>defaultSecurityType</code> property.
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	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>Policy</code> symbol is required, only application contexts that provide a <code>Policy</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> . 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> . In the <code>IDocumentTemplateDescriptor</code> interface, this is the <code>requiredSymbols</code> property.
availablesymbols	Required	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. In the <code>IDocumentTemplateDescriptor</code> interface, this is the <code>availableSymbols</code> property.

Form fields and field groups

To document templates that include Gosu expressions, you need to reference business data such as the current `Policy` 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 PolicyCenter data and fields in the document template. For example, to merge the policy number into the form field `PolicyInfo`, use the following expression.

```
<FormField name="PolicyInfo">Policy.CorePolicyNumber</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 policy by using the policy symbol. For example, get properties or call methods with code such as `policy.MyProperty` or `policy.myMethod()`.

`<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 `Policy.getPolicyInfoSummary`. Reference that function in a form field as follows.

```
<FormField name="PolicyInfoSummary">Policy.getPolicyInfoSummary()</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 policy information is” and a period character as the suffix, specify the field as shown below.

```
<FormField name="MainContactName"
  prefix="The policy information is "
  suffix=".">
  Policy.getPolicyInformation()
</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 <code><DisplayValues></code>	Description
<code><NullDisplayValue></code>	Customizes the display of <code>null</code> values. The content of the element is the value to substitute.

Child element of <code><DisplayValues></code>	Description
	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 <code><NullDisplayValue></code> to display something better if any part of the a field path expression is null. PolicyCenter also uses the <code>NullDisplayValue</code> if an invalid array index is encountered. For example, the expression "MyEntity.doctor[0].DisplayName" results in displaying the <code>NullDisplayValue</code> if the doctor array is empty. The null safety does not work with typical method calls on a null expression. For example, the expression <code>Obj.SubObject.Field1()</code> throws an exception if <code>Obj</code> is null.
<code><TrueDisplayValue></code>	Customizes the display of true values. The content of the element is the value to substitute.
<code><FalseDisplayValue></code>	Customizes the display of false values. The content of the element is the value to substitute.
<code><NumberFormat></code>	Customizes the display of numeric values.
<code><DateFormat></code>	Customizes the display of date values. If you provide a <code><DateFormat></code> child of <code><DisplayValues></code> , do not provide a <code><TimeFormat></code> .
<code><TimeFormat></code>	Customizes the display of time values. If you provide a <code><TimeFormat></code> child of <code><DisplayValues></code> , do not provide a <code><DateFormat></code> .

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
<code>MMM d, yyyy</code>	Jun 3, 2018
<code>MMMM d, yyyy</code>	June 3, 2018

Note: Four M characters specify the entire month name.

Date format	Example
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, PolicyCenter 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 policy acknowledgment letter to the main contact on the policy. Without context objects, each `FormField` element would have to repeat a prefix many times.

```
<FormField name="ToName">Policy.MainAccountContact.DisplayName</FormField>
<FormField name="ToCity">Policy.MainAccountContact.PrimaryAddress.City</FormField>
<FormField name="ToState">Policy.MainAccountContact.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>Policy.MainAccountContact</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 PolicyCenter 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>Policy.MainAccountContact</DefaultObjectValue>
  <PossibleObjectValues>Policy.getMyCustomRelatedContactsExtensionMethod()</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 policy as alternatives.

If you select a document template from the chooser, PolicyCenter 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
<code>EntityName</code>	The name of a PolicyCenter entity type such as <code>Policy</code> or <code>Activity</code> . 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", PolicyCenter displays a user picker.
<code>Bean</code>	To indicate support for any keyable entity, provide the literal text value "Bean". This is useful for heterogeneous lists of objects.
<code>TypeListName</code>	The name of a PolicyCenter typelist, such as "YesNo".
<code>string</code>	This value specifies a text value that appears in the user interface as a single line of text. The <code><DefaultObjectValue></code> tag must indicate the default text for this context object. This context object type always ignores the <code><PossibleObjectValues></code> tag. To use this context object type, the type value <code>string</code> must be all lower case.
<code>text</code>	Appears in the user interface as several lines of text. The <code><DefaultObjectValue></code> tag must be present, and its contents indicates the default text for this context object. If you use this, PolicyCenter ignores the <code><PossibleObjectValues></code> tag. To use this context object type, the type value <code>text</code> must be all lower case.
<code>date</code>	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`, `Policy`, or any other PolicyCenter 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 PolicyCenter 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 "Policy".
- `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. PolicyCenter 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 PolicyCenter 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 236

Template descriptor fields related to form fields and values to merge

Form fields dictate a mapping between PolicyCenter data and the merge fields in the document template.

For example, you might want to merge the policy number into a document field using the simple Gosu expression “`Policy.CorePolicyNumber`”.

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. PolicyCenter 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>Policy.MainAccountContact</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>
</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 `DocumentTemplateDescriptorXSDLocation` 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 `DocumentTemplateDescriptorXSDLocation` 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: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 PolicyCenter creates the merged document, the application tries to match attributes on the document template with properties on the Document entity. For each matched property, PolicyCenter 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: <code><%=FORM_FIELD_NAME%></code></p> <p>For example, for a form field called <code>PolicyNumber</code>:</p> <code><%=PolicyNumber%></code>
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 <code>InsuredName</code> and <code>PolicyNumber</code>. Your Word file must contain form fields called <code>InsuredName</code> and <code>PolicyNumber</code>.</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 PolicyCenter. 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 <code>InsuredName</code> and <code>PolicyNumber</code>. 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> <pre>InsuredName,PolicyNumber FakeName,FakePolicy</pre> <ol style="list-style-type: none"> 1. Start the Mail Merge wizard or chose Step by Step Merge Wizard. 2. When the wizard user interface asks about using an existing list, click Browse... 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 <code>InsuredName</code> or <code>PolicyNumber</code>.
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, PolicyCenter keeps a set of document templates on the PolicyCenter application server. The document templates are initially in the following directory:

`PolicyCenter/modules/configuration/config/resources/doctemplates`

At run time, the document templates are typically in the following directory.

`SERVER/webapps/pc/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 PolicyCenter.
The template descriptor	Template name with the suffix .descriptor	Test.doc.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.

```
PolicyCenter/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 109.

Document template descriptor optional cache

By default, PolicyCenter 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 PolicyCenter 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

PolicyCenter 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 e-mails 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 policy by setting `Document.Policy`.

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 to an external document management system, the externally-stored document is orphaned. The document exists in the external system but no PolicyCenter 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 = myPolicyPeriod.PrimaryNamedInsured
var fromContact : Contact = myPolicy.AssignedUser.Contact.Person
var subject : String = "Email Subject"
var body : String = "Email Body"

// Next, actually *send* the email
gw.api.email.EmailUtil.sendEmailWithBody(myPolicy, toContact, fromContact, subject, body)

// Next, create the document that records the email
var document : Document = new Document(myPolicy)
document.Policy = myPolicy
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("Policy", myPolicy)

// 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 PolicyCenter transaction rolls back, there is no stored reference in the PolicyCenter 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 PolicyCenter 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 PolicyCenter data regenerates a new, duplicate version of the document.

For the common case of validation errors, PolicyCenter avoids this problem. If a validatable entity fails validation, PolicyCenter saves the document UID in local memory. If the user fixes the validation error in that user session, PolicyCenter 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 PolicyCenter 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 PolicyCenter 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, ContactManager supports 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 PolicyCenter 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.

Note: The base configuration of PolicyCenter does not provide any screens that use the geocoding plugin for routing services or reverse geocoding

How PolicyCenter uses geocode data

The base configuration of PolicyCenter uses geocode data on location addresses to enable you to search for nearby locations and assemble them into reinsurable location groups. This geocoding feature of PolicyCenter is available only if your license for PolicyCenter includes use of Guidewire Reinsurance Management.

See also

- “Reinsurance integration” on page 431

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 PolicyCenter, calling a `GeocodePlugin` method is always synchronous. The caller waits until the method completes. For user-initiated requests, such as searching for nearby locations, the user interface blocks until the plugin responds.

PolicyCenter 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.

See also

- *System Administration Guide*

Using a proxy server with the geocoding plugin

To prevent PolicyCenter 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 691

Batch geocoding only some addresses

The `Address` entity has a property called `BatchGeocode`. The `Geocode` writer in PolicyCenter 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`.
- PolicyCenter verifies the following:
 - The address is an `AccountLocation` or subtype.
 - The address is a primary address of a `UserContact`.
 - The `BatchGeocode` property is `true` and the `GeocodeStatus` property is `none`.

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.

Built-in Bing maps geocoding plugin

PolicyCenter 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:

- One accepts `AbstractGeocodePlugin.LatLong` objects for the start and end waypoints.
- One accepts `Address` entities.

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.

See also

- [System Administration Guide](#)

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

There are general utility methods defined in the Gosu classes `GeocodingUtil` and `RoutingUtil` that 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 to be used in the `Context` `get` or `post` methods (not implemented) to create the associated `PendingResult` object from the request. Upon a `PendingResult execute`, the actual request to Bing Maps is performed, and the response is then processed. This method can only be run once per request.

There is a `RequestHandler` and associated `PendingResult` implementation provided, which is being 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.

There are default `Config` objects 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. If your license for PolicyCenter includes use of Reinsurance Management and you want to support searching for nearby locations, 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. Enable the user interface for geocoding features by configuring parameters in the `config.xml` file:

- ol style="list-style-type: none;">- a. In the **Project** window in Studio, navigate to **Configuration**→**config**, and then open the `config.xml` file.
- b. For PolicyCenter, modify the `UseGeocodingInPrimaryApp` parameter. This parameter specifies whether PolicyCenter displays the user interface to search for nearby locations. For example:

```
<param name="UseGeocodingInPrimaryApp" value="true"/>
```

See also

- *System 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	Re- quired
Geocode an address	<code>geocodeAddressBestMatch</code>	•
List possible matches for an address	<code>geocodeAddressWithCorrections</code> <code>pluginSupportsCorrections</code>	
Retrieve driving directions between two addresses	<code>getDrivingDirections</code> <code>pluginSupportsDrivingDirections</code> <code>pluginReturnsOverviewMapWithDrivingDirections</code> <code>pluginReturnsStepByStepMapsWithDrivingDirections</code>	
Retrieve a map for an address	<code>getMapForAddress</code> <code>pluginSupportsMappingByAddress</code>	
Retrieve an address from a pair of geocode coordinates	<code>getAddressByGeocodeBestMatch</code> <code>pluginSupportsFindByGeocode</code>	
List possible addresses from a pair of geocode coordinates	<code>getAddressByGeocode</code> <code>pluginSupportsFindByGeocodeMultiple</code>	

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 PolicyCenter 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 PolicyCenter needs to geocode an address immediately, PolicyCenter calls one of the geocoding plugin methods. For example, to get the best match for a geocoding request, PolicyCenter 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. PolicyCenter 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

PolicyCenter geocodes addresses in the background by using batch processes that call the geocoding plugin.

See also

- *System Administration Guide*

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. PolicyCenter 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 PolicyCenter 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

To support driving directions, implement 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.

PolicyCenter 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 `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

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.

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
<code>none</code>	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, PolicyCenter sets the address to this status, which indicates that the address has not been geocoded since it was last modified.
<code>failure</code>	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.
<code>exact</code>	This address was geocoded successfully, and the service provider indicates that the supplied geocode coordinates match exactly the complete address.
<code>street</code>	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.
<code>postalcode</code>	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.
<code>city</code>	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.

Encryption integration

Some data model properties can be encrypted before PolicyCenter 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. PolicyCenter 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 PolicyCenter.

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 PolicyCenter 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.

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. PolicyCenter 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. PolicyCenter 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.

- `Policy.SecretVal` and `Policy.SecretVal2` are encrypted properties.
- `Policy.ClearVal` and `Policy.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(Policy#SecretVal, Equals, "123")
query.compare(Policy#SecretVal, NotEquals, "123")
query.compare(Policy#SecretVal, Equals, tempStringValue)
query.compare(Policy#SecretVal, NotEquals, tempStringValue)

// Comparison of an encrypted column to another encrypted column
query.compare(Policy#SecretVal, Equals, q.getColumnRef("SecretVal2"))
query.compare(Policy#SecretVal, NotEquals, q.getColumnRef("SecretVal2"))

// Comparison of plaintext column with other plaintext values
query.compare(Policy#ClearVal, Equals, q.getColumnRef("ClearVal2"))
query.compare(Policy#ClearVal, NotEquals, q.getColumnRef("OtherVal2"))
```

The following comparisons are not valid. Although the SQL command that PolicyCenter 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(Policy#SecretVal, Equals, q.getColumnRef("ClearVal"))

// Comparison of a plaintext column to an encrypted column
query.compare(Policy#ClearVal, NotEquals, q.getColumnRef("SecretVal"))

// Comparison of an encrypted column using a relative operator
query.compare(Policy#SecretVal, GreaterThan, "123")
```

The following ordering request is not valid. Although the SQL command that PolicyCenter sends to the database is syntactically correct, sorting rows by an encrypted column is not meaningful.

```
query.select().orderBy(QuerySelectColumns.path(Paths.make(Policy#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.

- PolicyCenter and BillingCenter
- 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 257
- [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. PolicyCenter 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 PolicyCenter 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 PolicyCenter 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. PolicyCenter 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 PolicyCenter 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

PolicyCenter 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.

Name:	DESEncryptionSHA1															
Interface:	IEncryption															
+ -																
Gosu [gw.plugin.encryption.impl.PBEEncryptionPlugin]																
Gosu Class:	gw.plugin.encryption.impl.PBEEncryptionPlugin															
Environment:	<input type="text"/>															
Server:	<input type="text"/>															
<input type="checkbox"/> Disabled																
Parameters: <table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th>Name</th> <th>Value</th> <th>Envir</th> </tr> </thead> <tbody> <tr> <td>PBE.digest</td> <td>SHA1</td> <td></td> </tr> <tr> <td>PBE.encryption</td> <td>DESede</td> <td></td> </tr> <tr> <td>key</td> <td>This is some long ask string being used as a pass phrase</td> <td></td> </tr> <tr> <td>identifier</td> <td>DESEncryption2</td> <td></td> </tr> </tbody> </table>		Name	Value	Envir	PBE.digest	SHA1		PBE.encryption	DESede		key	This is some long ask string being used as a pass phrase		identifier	DESEncryption2	
Name	Value	Envir														
PBE.digest	SHA1															
PBE.encryption	DESede															
key	This is some long ask string being used as a pass phrase															
identifier	DESEncryption2															

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.

Name:	DESEncryptionSHA1															
Interface:	IEncryption															
+ -																
Gosu [gw.plugin.encryption.impl.PBEEncryptionPlugin]																
Gosu Class:	gw.plugin.encryption.impl.PBEEncryptionPlugin															
Environment:	<input type="text"/>															
Server:	<input type="text"/>															
<input type="checkbox"/> Disabled																
Parameters: <table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th>Name</th> <th>Value</th> <th>Envir</th> </tr> </thead> <tbody> <tr> <td>PBE.digest</td> <td>MD5</td> <td></td> </tr> <tr> <td>PBE.encryption</td> <td>DES</td> <td></td> </tr> <tr> <td>key</td> <td>This is some long ask string being used as a pass phrase</td> <td></td> </tr> <tr> <td>identifier</td> <td>DESEncryption1</td> <td></td> </tr> </tbody> </table>		Name	Value	Envir	PBE.digest	MD5		PBE.encryption	DES		key	This is some long ask string being used as a pass phrase		identifier	DESEncryption1	
Name	Value	Envir														
PBE.digest	MD5															
PBE.encryption	DES															
key	This is some long ask string being used as a pass phrase															
identifier	DESEncryption1															

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.

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.

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. PolicyCenter uses these plugins to decrypt any encrypted data in archived policies.

6. Start the server.

Result

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 or the encryption ID.
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.

Management integration

The management integration API enables external applications to trigger PolicyCenter actions or retrieve PolicyCenter data. For example, a console application can call the API to trigger PolicyCenter 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 PolicyCenter configuration parameters. The following types of PolicyCenter 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, PolicyCenter 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 PolicyCenter, 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 PolicyCenter to communicate with multiple external applications. Each API implementation must have a unique name.

PolicyCenter exposes data through both its user interface and the management integration API.

In the PolicyCenter **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 PolicyCenter 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 PolicyCenter 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 `java-api` directory after running the following command.

```
gwb genJavaApi
```

Extract the files in the archive file. The source files for the example plugins will be located in the following PolicyCenter directory.

```
java-api/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 PolicyCenter 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 `JMXRMICollector` 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 topics.

Register the `JMXManagementPlugin`

1. Copy the contents of the `java-api/examples/src/examples/pl/plugins/management` directory to the `PolicyCenter modules/configuration/src/examples/pl/plugins/management` directory. This operation places the example plugin source files into the Guidewire Studio Project structure.
2. In the Studio Project window, select **configuration**→**config**→**Plugins**→**registry**.
3. Right-click the **registry** directory and select **New**→**Plugin** to show the **Plugin** dialog window.
4. Enter a plugin **Name** of `JMXManagementPlugin`. Enter the **Interface** name of `ManagementPlugin`. Click **Okay** to create the plugin and show its initial settings in the Plugin Registry.
5. Click the green Plus icon and select **Add Java Plugin**.
6. In the **Java Class** field, browse for the `JMXManagementPlugin` and select it.

In addition, the example support files used by the `JMXManagementPlugin` must be combined into a JAR file. Use any Java IDE to create the JAR file. Copy the generated JAR file to the `PolicyCenter modules/configuration/plugins/basic/lib` directory.

Call the `JMXManagementPlugin` from an external application

The following example Java code demonstrates how an external application can retrieve a PolicyCenter 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) {  
    e.printStackTrace();  
}  
}
```

Other plugin interfaces

Plugins are software modules that PolicyCenter 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 PolicyCenter 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 PolicyCenter 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 PolicyCenter 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, PolicyCenter calls the `IPreUpdateHandler` plugin whenever it commits a bundle to the database. PolicyCenter executes the plugin first before applying any existing preupdate rule set, thereby enabling preprocessing of the objects in the bundle.

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

PolicyCenter provides a default implementation of the `IPreUpdateHandler` plugin in Gosu class `PreUpdateHandlerImpl`, 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 `executePreUpdate` method on the plugin examines each added or modified object in the bundle and retains any examined object that meets the following criteria:

- The object is of the type `AccountUserRoleAssignment`. The method logic retains the `Account` portion of the object.
- The object is of the type `JobUserRoleAssignment`. The method logic retains the `Job` portion of the object.
- The object is of the type `EffDated`. If the bundle includes multiple objects with the same effective date, the method ensures that only a single instance is retained for a particular effective date.

After examining the bundle objects, the method executes the preupdate logic on the retained objects.

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 does not implement the Validation plugin. You can provide custom code to implement the plugin and then register it by using the standard procedure.

Work item priority plugin

Configure how PolicyCenter 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 PolicyCenter, 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 PolicyCenter, 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 PolicyCenter, 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.

JavaEmailMessageTransport class

The `JavaEmailMessageTransport` 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 `JavaEmailMessageTransport` 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 `defaultSenderId` and `defaultSenderIdAddress` 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 `defaultSenderId` and `defaultSenderIdAddress` 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 PolicyCenter to retrieve one or more email templates. To see the plugin registry, navigate in the **Project** window to **configuration**→**config**→**Plugins**→**registry**→`IEmailTemplateSource.gwp`.

The base configuration provides a demonstration plugin implementation class, `gw.plugin.email.impl.LocalEmailTemplateSource`, that is registered in `IEmailTemplateSource.gwp`. The `LocalEmailTemplateSource` class constructs an email template from files stored on the local file system. As a result, the sample implementation is for demonstration purposes and is unsuitable for use in a clustered environment. The class's `getEmailTemplates` method searches for an email template.

```
getEmailTemplates(locale : ILocale, valuesToMatch : Map) : IEmailTemplateDescriptor[]
```

The 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 <code>resource</code> → <code>emailtemplates</code> 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 PolicyCenter 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, PolicyCenter generates the most appropriate base URL, based on settings in `config.xml`.

In some cases, this behavior is inappropriate. For example, suppose you hide PolicyCenter 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 PolicyCenter 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 at the user starts with `http` instead of `https`. This breaks image loading and display because the browser tries to load the images relative to the `http` URL. The load balancer rejects the requests because they are insecure because they do not use HTTPS/SSL.

Avoid this problem by writing a custom `IBaseURLBuilder` base URL builder plugin and registering it with the system.

You can base your implementation on the built-in example implementation found at the following path.

```
PolicyCenter/java-api/examples/src/examples/plugins/baseurlbuilder
```

To handle the load balancer case mentioned earlier, the base URL builder plugin can look at the HTTP request's header. If a property that you designate exists to indicate that the request came from the load balancer, return a base URL with the prefix `https` instead of `http`.

The built-in plugin implementation provides a parameter `FqdnForUrlRewrite` which 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 PolicyCenter 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. If the `FqdnForUrlRewrite` parameter is not set, the end user is responsible for entering a URL with a fully qualified domain name.

There is another parameter called `auto` which tries to auto-configure the domain name. This setting is not recommended for clustering environments. For example, do not use this if the web server and application server are not on the same machine, or if multiple virtual hosts live in the same machine. In these cases, it is unlikely for the plugin to figure out the fully qualified domain name automatically.

In Project window in Studio, navigate to **configuration**→**config**→**Plugins**→**registry** and the open **IBaseURLBuilder**. Add the parameter `FqdnForUrlRewrite` with the value of your domain name, such as "`mycompany.com`". The domain name must specify the Fully Qualified Domain Name to be enforced in the URL. If the value is set to "`auto`", the default plugin implementation makes the best effort to calculate the server FQDN from the underlying configuration.

If PolicyCenter 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, PolicyCenter generates the most appropriate base URL, based on settings in `config.xml`.

In some cases, this behavior is inappropriate. For example, suppose you hide PolicyCenter 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 PolicyCenter 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`. This breaks image loading and image display 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.

This situation can be avoided by writing a custom plugin that creates an appropriate base URL. Requests that originate from the load balancer can then be forced to return a base URL with a scheme of `https` instead of `http`. The custom plugin implements the **IBaseURLBuilder** interface and is enabled by registering it with PolicyCenter.

The custom 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`.

An example implementation of the **IBaseURLBuilder** plugin is provided in the source file shown below. The source file is created as part of the operation of generating the Java API libraries and then unzipping the resulting `PolicyCenter/java-api/java-examples.zip` file.

```
PolicyCenter/java-api/examples/src/examples/pl/plugins/baseurlbuilder/ExampleBaseURLBuilder.java
```

To enable the custom plugin, it must be registered with PolicyCenter. In the Studio **Project** window, navigate to **configuration**→**config**→**plugin**→**registry**. Right-click **registry** and select **New**→**Plugin**.

Vehicle identification number plugin

The `IVinPlugin` plugin is a simple plugin that returns vehicle information from a standard vehicle identification number (VIN). The single method—`getVehicleInfo`—takes a VIN as a `String` and returns a `VinResult` object, which contains the specific vehicle's make, model, year, and color.

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

PolicyCenter 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.

PolicyCenter 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.api.phone.DefaultPhoneNormalizerPlugin`.

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 PolicyCenter 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**→**policy** 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.PatternReplaceFilterFactory"
            pattern= "[^a-zA-Z0-9 ]" replacement= "" replace= "all"/>
        <filter class="solr.TrimFilterFactory"/>
        <filter class="com.guidewire.solr.analyzers.phone.PhoneFilterFactory"
            region="US"
            parsingExtensionLength="10"/>
    </analyzer>
    <analyzer type="query">
        <tokenizer class="solr.KeywordTokenizerFactory"/>
        <filter class="solr.PatternReplaceFilterFactory"
            pattern= "[^a-zA-Z0-9 ]" replacement= "" replace= "all"/>
        <filter class="solr.TrimFilterFactory"/>
        <filter class="com.guidewire.solr.analyzers.phone.PhoneQueryFilterFactory"
            region="US"
            parsingExtensionLength="10"/>
    </analyzer>
</fieldType>
```

See also

-

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 PolicyCenter 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.

A problem arises in determining which official ID in the `OfficialIDs` array to store in the `TaxID` field for a given contact. Different types of contacts have different types of official IDs. In the U.S. for example, a human being's

official Social Security Number (SSN) is used as the person's tax ID. In the U.S. furthermore, a company's official Federal Employer Identification Number (FEIN) is used as the company's tax ID.

You configure PolicyCenter with official ID types in the `OfficialID` typelist. For example, the base configuration includes type keys for the U.S. Social Security Number and the U.S. Federal Employer Identification Number. 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.

Use the default Gosu implementation of the `OfficialIdToTaxIdMappingPlugin` interface to configure which typekeys from the `OfficialID` typelist PolicyCenter treats 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 PolicyCenter to treat that official ID type as a tax ID. The default implementation that PolicyCenter provides returns `true` for the official ID typekeys that have SSN or FEIN as codes.

Testing clock plugin (for non-production servers only)

The PolicyCenter time can be modified from the user interface or programmatically with the `ITestingClock` plugin. The ability to change the time is provided for testing PolicyCenter behavior over a long, simulated period of time. Changing the time is supported only for testing purposes on non-production development servers. Do not manually change the time or call the plugin on production servers.

The plugin interface has two methods—`getCurrentTime` and `setCurrentTime`—which get and set the current time using the standard PolicyCenter format of milliseconds stored in a `long` integer.

If you cannot set the time in the `setCurrentTime` function—for example if you are using an external time server, and it is temporarily unreachable—throw exception `java.lang.IllegalArgumentException`.

Time must always advance and never go backward. Setting the testing clock to go back in time is not supported and causes unpredictable behavior.

After the application calls `setCurrentTime`, the time advances for any code that calls the `DateUtil` utility class to get the current date. The `setCurrentTime` method affects all PolicyCenter 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 get the date with `gw.api.util.DateUtil.CurrentDate` rather than other APIs. For example, the standard Java class `java.util.Date` method will return the actual server time and not the PolicyCenter time.

Certain areas in PolicyCenter 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

Using the testing clock

The testing clock is available from the PolicyCenter 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
NOTE Only available when server is in development mode. -->
<param name="EnableInternalDebugTools" value="true"/>
```

7. Save your changes.
8. Stop, and then restart PolicyCenter.

Access the testing clock from the user interface

Procedure

1. If you are operating a cluster of PolicyCenter 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. To advance the clock using a picker, type **Alt+Shift+T**, and then click the **Internal Tools** tab. In the sidebar, click **Testing System Clock**. Set the clock as needed by clicking one of the **Add** buttons or entering a specific date and time. Finally, click **Change Date**.
5. Start up the other servers.

Use the testing clock in a cluster environment

Procedure

1. Stop all servers in the cluster, until only a single server is running. If you wrote your own testing clock implementation that does not require a running server then shut down all servers in the cluster.
2. Advance the testing clock. If you wrote your own implementation, do whatever is necessary to work with your own private implementation. For example, you might make SQL modifications of some parameter or adjust some offset encoded in a configuration file.
3. Restart all servers in the cluster.

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 99.

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 PolicyCenter 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

PolicyCenter provides a plugin interface that supports high performance multi-threaded processing of inbound data. PolicyCenter 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. PolicyCenter 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 PolicyCenter 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

PolicyCenter provides the multi-threaded inbound integration system in plugin interfaces for two use cases. Each interface defines a contract between PolicyCenter 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, PolicyCenter uses the value of the `stoponerror` parameter in the configuration file to determine what action to take. If that value is `true`, PolicyCenter performs error handling that is applicable to the integration type. If the value of `stoponerror` is not `true`, PolicyCenter 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.

PolicyCenter 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.

PolicyCenter 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 PolicyCenter 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`. PolicyCenter 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. PolicyCenter 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. PolicyCenter 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 298
- “Registering a plugin implementation class” on page 136

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 PolicyCenter, 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 PolicyCenter, the `<integrations>` element contains a list of subelements with pre-defined inbound integration names.

See also

- “Thread pool configuration XML elements” on page 286
- “Inbound integration configuration XML elements” on page 287
- “Registering a plugin implementation class” on page 136

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	Re-required	Default value	Description
name	string	•		A unique identifying name for this thread pool in the <code>inbound-integration-config.xml</code> file. The <code>name</code> 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

PolicyCenter 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`, PolicyCenter does not enable the thread pool or integration.

If the value of that attribute is `false`, the action that PolicyCenter takes depends on the value of the optional `env` attribute. If you provide no value for this attribute, PolicyCenter 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, PolicyCenter 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	Re-required	Default value	Description
name	string	•		A unique identifying name for this inbound integration in the <code>inbound-integration-config.xml</code> file. 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.
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	Re-required	De-default value	Description
threadpool	string	•		The unique name of a thread pool as configured earlier in the file. 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> .
pollinginterval	string	•	60	The time interval in seconds between polls, though the algorithm interacts with the <code>throttleinterval</code> and the <code>ordered</code> parameter.
throttleinterval	string	•	60	The time interval in seconds after polling, though the algorithm interacts with the <code>pollinginterval</code> and the <code>ordered</code> parameter.
ordered	string	•	true	By default, the inbound file integration handles files in a single thread sequentially. 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. 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.
stoponerror	string	•	true	If <code>true</code> , PolicyCenter 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. If not <code>true</code> , PolicyCenter skips the item if an error occurs. Be sure to log any errors or notify an administrator.
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	Re-required	Default value	Description
traceabilityidcreationpoint	string	•	INBOUND_INTEGRATION_FILE	The value is specified by the constants in <code>gw.logging.TraceabilityIDCreationPoint</code> . 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 <code>InboundIntegrationHandlerPlugin</code> plugin interface. The value is the <code>.gwp</code> file name, not the implementation class name.
processingmode	string	•	line	To process one line at a time, set to <code>line</code> . In your handler class in the <code>process</code> method, you must downcast to <code>String</code> . To process one file at a time, set to <code>file</code> . In your handler class in the <code>process</code> method, you must downcast to <code>java.nio.file.Path</code> .
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 <code>true</code> , PolicyCenter 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 <code>false</code> , PolicyCenter requires that all of these directories must already exist. If any directories do not already exist, the server does not start. For better security, set to <code>false</code> .

<jms-integration> XML element subelements

The JMS inbound integration XML element has the following additional subelements, in the order shown.

Subelement name	Type	Re-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: • 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.
durablesubscription	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.
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	Re-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 setup method. These parameters are in the <code>java.util.Map</code> that PolicyCenter 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:

- 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.
- PolicyCenter 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.
- 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 PolicyCenter includes classes that support file-based input with high-performance multi-threaded processing. PolicyCenter 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`, PolicyCenter sends one line at a time to the handler as a `String` object. If that subelement has the value `file`, PolicyCenter 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

PolicyCenter 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 PolicyCenter 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. PolicyCenter 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.

See also

- “Inbound integration handlers for file and JMS integrations” on page 284
- “Inbound integration configuration XML elements” on page 287
- “Example of an inbound file integration” on page 294

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>
  <pluginhandlers>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 292

Inbound JMS integration

The base configuration of PolicyCenter includes a high-performance multi-threaded integration with inbound queues of Java Message Service (JMS) messages. PolicyCenter 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.

PolicyCenter logs any exception that the integration `process` method in your code throws. The next actions that PolicyCenter takes depend on the `stoponerror` and `ordered` configuration subelements.

- If both the `stoponerror` and `ordered` subelements have the value `true`, PolicyCenter 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`, PolicyCenter 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.

PolicyCenter can use JMS implementations on the application server but PolicyCenter 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 PolicyCenter application, deploy PolicyCenter on one of the other supported application servers. For additional advice on setting up or configuring an inbound JMS integration with PolicyCenter, contact Guidewire Customer Support.

See also

- “Using the inbound integration polling and throttle intervals” on page 291

Create an inbound JMS integration

PolicyCenter 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 PolicyCenter 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. PolicyCenter 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>**.

You can use a **<jms-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.

- 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.

See also

- “Inbound integration handlers for file and JMS integrations” on page 284
- “Inbound integration configuration XML elements” on page 287
- “Example of an inbound JMS integration” on page 297

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/>
  <durablesSubscription/>
  <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_ARRIVAL=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 295

Custom inbound integrations

The base configuration of PolicyCenter 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 PolicyCenter.

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 275

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 <code>gw.api.integration.inbound</code> .	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 <code>Factory</code> .

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 <code>gw.api.integration.inbound</code> .	For each polling interval, the factory instantiates the class that implements the interface <code>WorkSetProcessor</code> .
A work set processor	WorkSetProcessor in package <code>gw.api.integration.inbound.work</code> . If your plugin supports the optional feature of being transactional, implement the subinterface <code>TransactionalWorkSetProcessor</code> .	The work set processor acquires and allocates resources. This class also instantiates the class that implements the interface <code>Inbound</code> . The main method for processing one unit of work in a <code>WorkData</code> object is the <code>process</code> method in this class.

Class that you write	Interface to implement	Description
A class to find work	Inbound in package gw.api.integration.inbound.work.	The inbound class defines how to find work in its <code>findWork</code> method that returns new work data sets. This class also instantiates the class that implements the interface <code>WorkDataSet</code> .

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 (<code>WorkData</code>) objects and any necessary context information to operate on the data. This <code>WorkDataSet</code> object also instantiates the class that implements the <code>WorkData</code> 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. PolicyCenter 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. PolicyCenter 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

PolicyCenter 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, PolicyCenter behavior depends on the `stoponerror` and `ordered` configuration parameters. If `stoponerror` is `true`, in the following cases, PolicyCenter 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`, PolicyCenter 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 291

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

PolicyCenter 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.

See also

- “Inbound integration configuration XML elements” on page 287

Part 4

Messaging

Messaging and events

You can send messages to external systems after something changes in PolicyCenter, such as a changed policy. The changes trigger events, which trigger your code that sends messages to external systems. For example, if you submit a new policy, your messaging code notifies a billing system about the new policy.

PolicyCenter 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. PolicyCenter queues the messages and then dispatches them to the appropriate external systems.

This topic explains how PolicyCenter generates messages in response to events and how to connect external systems to receive those messages.

Overview of messaging

Event

An event is an abstract notification of a change in PolicyCenter 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 PolicyCenter a new policy submission triggers an event. The event name is "`IssueSubmitted`".

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. PolicyCenter 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 to send messages to.

Generally speaking, a messaging destination represents a single external system. Register a destination for each external system, even if the system is used for multiple types of data.

Messaging destinations are registered in Studio.

Each destination can register a list of event names for which it wishes 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 desired.

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 specific entity types are supported for each Guidewire application.

Determining which primary object, if any, applies for a messaging destination is critical to understanding how PolicyCenter orders messages.

- The default primary entity is `Account`. Most objects are associated with an account indirectly as subobjects of `Policy` or `PolicyPeriod`, or associated directly with an account.
- The `Contact` entity is the recommended alternative primary entity for a messaging destination. The alternative primary entity applies just to that messaging destination.
- If required, you can specify the `Policy` or `PolicyPeriod` entity as the alternative primary entity for a messaging destination. However, messages on these alternative entities are not safe-ordered at the `Account` level. Therefore, you must be sure that downstream systems can handle out-of-order messages on the `Account`. For information on safe-ordering, see “[Sending safe-ordered messages](#)” on page 322.

Some objects are not associated with any primary object. For example, `User` objects are not associated with a single policy. 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.

Acknowledgment

An acknowledgment is a formal response from an external system to PolicyCenter 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.

PolicyCenter 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. PolicyCenter will try multiple times to resend the message.
- A NAK error reported from the external system. PolicyCenter 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 PolicyCenter is the Account. Accordingly, messages are grouped based on their associated Account 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 object 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

PolicyCenter does not assume any specific type of transport or message formatting.

Destinations deliver 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, such as a message queue.
- Save to special files in the file system – For large-scale batch handling, you could send a message by implementing writing data to local text files that are read by nightly batch processes. If you do this, remember to make your plugins thread-safe when writing to the files.
- Send e-mails – The destination might send e-mails, 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.

1. Application startup – At application startup, PolicyCenter checks its configuration information and constructs messaging destinations. Each destination registers for specific events for which it wants notifications.
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 – PolicyCenter 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 `IssueSubmission`. When such an event is identified, the rules might generate a message with an XML payload that describes information about the submission.

4. PolicyCenter sends message to a destination – Messages are put in a queue and handed one-by-one to the messaging destination.

In PolicyCenter, a message destination representing an external policy management system might take the XML payload and submit the message to a message queue. The message might notify the external system about a submission.

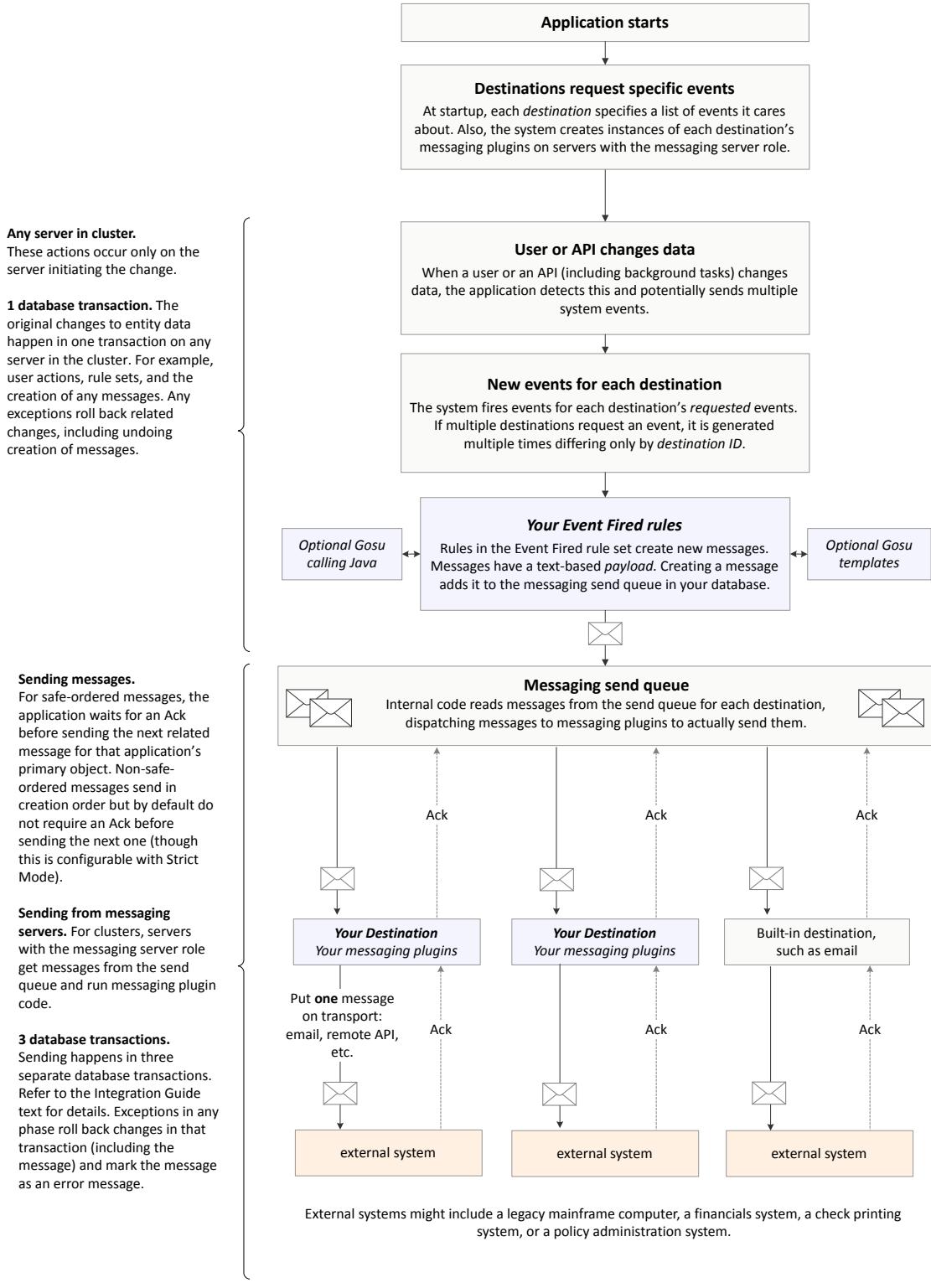
5. PolicyCenter waits for an acknowledgment – The external system replies with an acknowledgment to the destination after it processes the message, and the destination's messaging plugins process this information. If the message was successfully sent, the messaging plugins submit an ACK, and PolicyCenter 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 PolicyCenter application server starts, the application initializes all destinations. At system startup, PolicyCenter 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 PolicyCenter 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 PolicyCenter.

For example, PolicyCenter triggers an event if you add a policy or change a note.

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. PolicyCenter generates messaging events.

A single action might trigger more than one event. PolicyCenter checks whether each destination has listed each relevant event in its messaging configuration. For each messaging destination that listens for that event, PolicyCenter calls your Event Fired business rules. If multiple destinations want notifications for a specific event, PolicyCenter 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 PolicyCenter 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. PolicyCenter invokes Event Fired rules.

PolicyCenter 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, PolicyCenter 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, PolicyCenter dispatches messages to messaging destinations.

PolicyCenter retrieves messages from the send queue and dispatches messages to messaging plugins for each destination.

To send a message, PolicyCenter 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, PolicyCenter 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 PolicyCenter. 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. Also, where appropriate, you could advance a workflow.

8. After an ACK or NAK for a safe-ordered message, PolicyCenter 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, PolicyCenter 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 PolicyCenter 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 PolicyCenter 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> <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 <code>messaging</code> server role. If more than one server has the <code>messaging</code> 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>

Message lifecycle phase	Description
	<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>PolicyCenter includes multithreaded inbound integration APIs that you can optionally use in conjunction with MessageReply 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 InboundIntegrationMessageReply plugin. InboundIntegrationMessageReply is a subinterface of MessageReply.</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 plugin. 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 PolicyCenter 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 PolicyCenter 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 PolicyCenter 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, PolicyCenter 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, PolicyCenter 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, PolicyCenter 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. PolicyCenter 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 PolicyCenter 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 PolicyCenter 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`). PolicyCenter 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. PolicyCenter 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 PolicyCenter web service to acknowledge the message. PolicyCenter 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 policy as a document. The email APIs in the base configuration use the email destination provided in the base configuration. PolicyCenter always creates this email destination, independent of your configuration settings.

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, PolicyCenter 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, PolicyCenter 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`, PolicyCenter 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 `Account` 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.

- 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 send synchronously. A message acknowledgment must be received before PolicyCenter 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 account. PolicyCenter can send these ten unrelated messages immediately, rather than synchronously. However, if the send queue contains ten messages for the same account, PolicyCenter 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.

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:

- PolicyCenter 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.
- PolicyCenter runs both `send` and `afterSend` methods in a single database transaction and commits the changes, assuming that no exceptions occurred.
- PolicyCenter 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 PolicyCenter is <code>Policy</code> .
Primary object	A primary object is a specific instance of a primary entity, which PolicyCenter stores in table <code>pc_policy</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. PolicyCenter stores the activity root object in table <code>pc_activity</code> .
Message object	A message event generates a message object. PolicyCenter stores a message object instance in table <code>pc_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 `pc_message`. It also locks the direct (root) entity associated with the message, an instance of an activity object (in `pc_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 `pc_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 Policy triggers a message event. The message system then locks the primary object associated with the message, Policy in `pc_policy` 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:

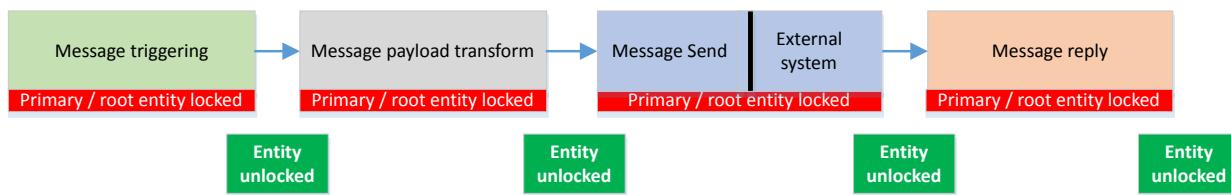
- While the `beforeSend` method executes
- While the `send` method executes
- While processing message-reply operations
- While marking a message as skipped

Locking and concurrent data exceptions

Message and primary entity locking operations exist outside the concurrent data exception system. PolicyCenter 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 pc_message for the message object
- The row in the database table for the instance of the direct object for the message, pc_activity for an activity object, for example.

Configuration parameter `LockPrimaryEntityDuringMessageHandling` affects this behavior:

<code>LockPrimaryEntityDuringMessageHandling</code>	Behavior
true	The messaging system locks the primary object of the message. For example, for a message triggered by an activity associated with a policy, the message system locks the appropriate row in table pc_policy.
false	The messaging system does not lock the primary object, but it does lock the table row for the message object itself (pc_message) 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.

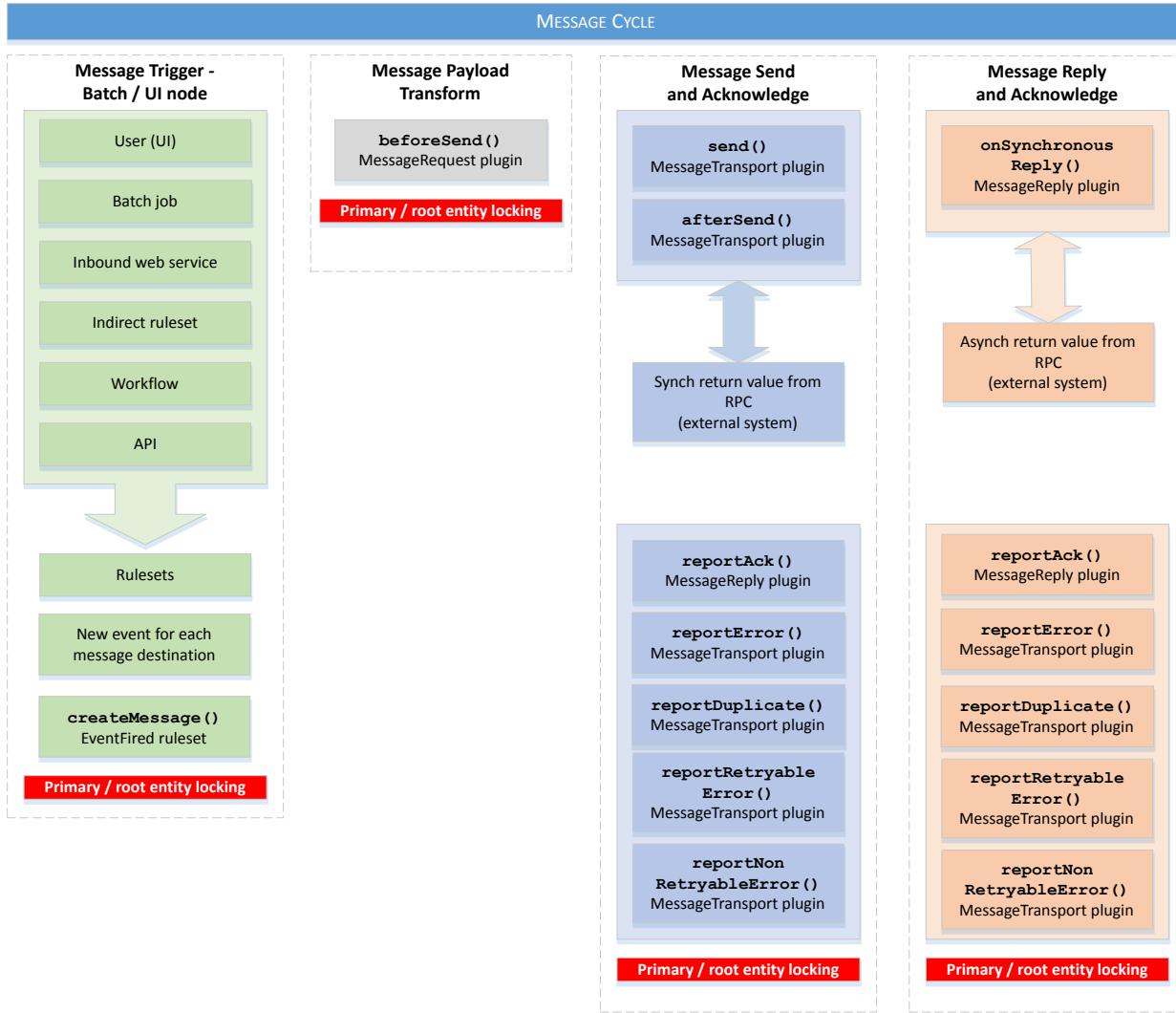
The intent of locking database rows is to prevent other transactions, users, or activities from modifying the content of the message during message processing. Locking the primary object ensures that only one message can affect the policy at a time.

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 PolicyCenter 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 PolicyCenter pc_Message and pc_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.	pc_Message.Creationtime
2. Payload transformation	pc_Message.BeforeSendLockTime (MessageRequest plugin)	pc_Message.BeforeSendLockedTime	pc_Message.BeforeSendTime This column has a value if the record is being queried. The value of this column is always NULL in pc_MessageHistory.
3. Message send	pc_Message.SendLockTime (MessageTransport plugin)	pc_Message.SendLockedTime	pc_Message.SendTime
4. Message after send	Inherits database lock in the send method (MessageTransport plugin)		pc_Message.AfterSendTime
5. Message retry	pc_Message.SendLockTime (MessageTransport plugin)		pc_Message.RetryTime (time in future to retry the message)
6. Message reply, acknowledgment	pc_Message.ackMessage (MessageReply plugin)		pc_MessageHistory.ackedMessage (time acknowledgment received)

In general, use the data in table pc_MessageHistory to perform your analysis. The message system populates the data in the pc_MessageHistory table after it sends out the message, moving the relevant data from the pc_Message table into the pc_MessageHistory table at that time.

Table column meanings

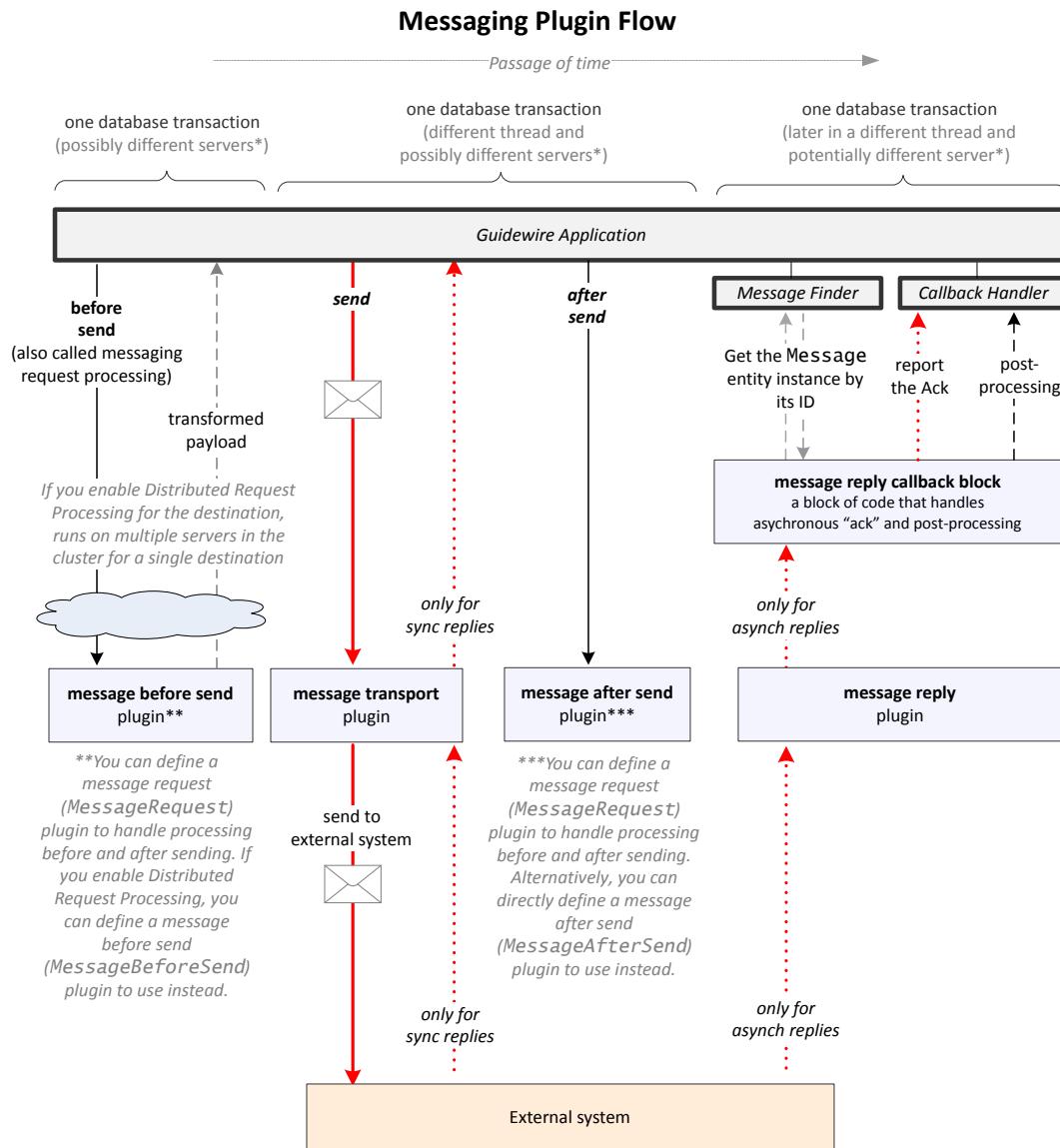
The following table describes the table columns listed in the previous table.

Column	Contains
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.
BeforeSendLockTime	The time when the MessageRequest plugin requests a lock on this message record for payload generation.
BeforeSendLockedTime	The time when 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.

Column	Contains
BeforeSendTime	The time between payload generation and updating the message record in the database. If there is an extensive amount of time spent at this step, it is due to volume of message, coding issues, or issues with external integration.
QueryTime	The time when the MessageTransport plugin queries the database for the record (pc_Message). In table pc_MessageHistory, this value is always NULL.
SendLockTime	The time when the MessageTransport plugin requests a lock of the message record for the Message.send method call.
SendLockedTime	The time when 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 after the Message.send method call completes and the message exits the MessageTransport plugin. If there is an extensive amount of time spent at this step, it is due to volume of message, coding issues, or issues with external integration.
AfterSendTime	The time after the Message.afterSend method call completes and the message exits the MessageTransport plugin.
AckedTime	The time after the Message.ackMessage method call completes and the reply message exits the MessageReply plugin.
RetryTime	Stores the time scheduled (in the future) to retry sending the message record.

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

Guidewire code
Code that you write
External system

- **Sending the message payload**
- **Reporting the message reply**

Messaging events in PolicyCenter

PolicyCenter 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 policy. 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 policy's producer's address? Was it an insured's address?

In PolicyCenter, most subobjects in an account have properties that you can use to access related objects. For example, most account subobjects have an **Account** property that points to the account. Most policy period subobjects have a **PolicyPeriod** property that points to the **PolicyPeriod** that includes this object. Also, remember that every policy belongs to exactly one account.

PolicyCenter 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 policy type on any policy generates a changed event on the policy.

The changes are about the change to that database row itself, not on subobjects. For example, PolicyCenter 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, switching to a different policy contact on a policy is a change to the policy. A change on the policy contact is a policy contact property change, but not a policy change.

List of messaging events

The following table describes the events that PolicyCenter raises. In this table, standard events refer to the added, changed, and removed events for entities that generate events. For example, the **Policy** 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 **PolicyAdded**, **PolicyChanged**, and **PolicyRemoved** events if one or more destinations registered for those event names.

Entity	Events	Description
Account	AccountAdded AccountChanged AccountRemoved	Standard events for root entity Account.
	PersonalDataPurge	An Account purge was committed as part of a personal data purge. The Event Fired rule PersonalDataPurge , in the EventMessage rule set category, creates a message in response to the firing of the PersonalDataPurge event. You must define your own transport and destination for the message.
	ResyncAccount	Resync an account. This is an administrator request to drop all pending and in error messages for an account, including all its policies. Then, your Event Fired rules try to resync the account with the external system to recover from integration problems. Administrators can request a resync from the application user interface or through the web services API.
Activity	ActivityAdded ActivityChanged ActivityRemoved	Standard events for root entity Activity. The ActivityChanged event triggers after an activity updates, including if a user marks it as completed or skipped.
Audit	JobAdded JobChanged JobRemoved	Standard events for root entity Job.
Cancellation	CancellationAdded CancellationChanged CancellationRemoved	Standard events for root entity Cancellation.

Entity	Events	Description
Document	DocumentAdded	Standard events for root entity Document. Some implementations do not let users remove documents once they have been added, so the remove event may not be called.
	DocumentChanged	
	DocumentRemoved	
DocumentStore		When the asynchronous document content storage code attempts to store in the database, PolicyCenter triggers a DocumentStore event on the relevant Document entity instance.
	FailedDocumentStore	When the asynchronous document content storage code fails to store in the database, PolicyCenter 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.
InboundHistory	InboundHistoryAdded	Standard events for root entity Invoice.
	InboundHistoryChanged	
	InboundHistoryRemoved	
Job	JobAdded	Jobs entities only exist as subtypes of Job, such as Audit. Those subtypes generate standard events for root entity Job.
	JobChanged	
	JobRemoved	
JobPurged		The job was removed through quote purging.
	RequestQuote	You can use this event to send a message to an external rating system to request a quote for a particular policy revision.
Letter	LetterAdded	Standard events for root entity Letter.
	LetterChanged	
	LetterRemoved	
Note	NoteAdded	Standard events for root entity Note.
	NoteChanged	
	NoteRemoved	
Organization	OrganizationAdded	Standard events for root entity Organization.
	OrganizationChanged	
	OrganizationRemoved	
Policy	PolicyAdded	Standard events for root entity Policy. Most events are not on the policy entity but on an individual policy period entity.
	PolicyChanged	
	PolicyRemoved	
PersonalDataPurge		A Policy purge was committed as part of a personal data purge. The EventFired rule PersonalDataPurge, in the EventMessage rule set category, creates a message in response to the firing of the PersonalDataPurge event. This message has no transport and destination, but you can define them or create your own rule.
	PolicyPurged	The policy period was removed through quote purging.
TransferPolicy		A policy transferred. This is a custom event used only by the PolicyCenter integration with BillingCenter.
	TransferPolicy	
PolicyChange	PolicyChangeAdded	Standard events for root entity PolicyChange.
	PolicyChangeChanged	
	PolicyChangeRemoved	

Entity	Events	Description
PolicyPeriod	BindSubmission	Fired to instruct an external system to bind a submission for a particular policy revision.
	CancelPeriod	A period needs to be canceled in the billing system. This is a custom event that is part of the PolicyCenter integration with BillingCenter.
	ChangePeriod	A period needs to be changed in the billing system. This is a custom event that is part of the PolicyCenter integration with BillingCenter.
	CreatePeriod	A new period needs to be sent to the billing system. This is a custom event that is part of the PolicyCenter integration with BillingCenter.
	FinalAudit	A period needs to be audited in the billing system. This is a custom event that is part of the PolicyCenter integration with BillingCenter.
	IssueCancellation	A message fired to notify an external system to issue a cancellation for a particular policy revision. The messaging plugin that submits the ACK typically advances workflow by finding the workflow entities and calls methods with names that begin with <code>finish</code> and <code>fail</code> . For example, call <code>Submission.finishIssue(PolicyPeriod)</code> . Or, you can call <code>workflowInvokeTrigger(WorkflowTriggerKey)</code> . Workflow actions during acknowledgment are optional. The workflow does not automatically move forward due to an acknowledgment.
	IssuePeriod	A period needs to be issued in the billing system. This is a custom event that is part of the PolicyCenter integration with BillingCenter.
	IssuePolicyChange	A message fired to notify an external system to issue a policy change for a particular policy revision. The messaging plugin that submits the ACK typically advances workflow by finding the workflow entities and calls methods with names that begin with <code>finish</code> and <code>fail</code> . For example, call <code>Submission.finishIssue(PolicyPeriod)</code> . Or, you can call <code>workflowInvokeTrigger(WorkflowTriggerKey)</code> . Workflow actions during acknowledgment are optional. The workflow does not automatically move forward due to an acknowledgment.
	IssueReinstatement	A message fired to notify an external system to issue a reinstatement for a particular policy revision. The messaging plugin that submits the acknowledgment typically advances the workflow by finding the workflow entities and calls their methods with names that begin with <code>finish</code> and <code>fail</code> . For example, call <code>Submission.finishIssue(PolicyPeriod)</code> . Or, you can call <code>workflowInvokeTrigger(WorkflowTriggerKey triggerKey)</code> . Workflow actions during acknowledgment are optional. The workflow does not automatically move forward due to an acknowledgment.

Entity	Events	Description
	IssueRenewal	A message fired to notify an external system to issue a renewal for a particular policy revision. The messaging plugin that submits the acknowledgment typically advances the workflow by finding the relevant workflow entities and calls their methods with names that begin with <code>finish</code> and <code>fail</code> . For example, call <code>Submission.finishIssue(PolicyPeriod)</code> . Or, you can call <code>workflowInvokeTrigger(WorkflowTriggerKey triggerKey)</code> . Workflow actions during acknowledgment are optional. The workflow does not automatically move forward due to an acknowledgment.
	IssueRewrite	A message fired to notify an external system that to issue a rewrite for a particular policy revision. The messaging plugin that submits the acknowledgment typically advances the workflow by finding the workflow entities and calls their methods with names that start with <code>finish</code> and <code>fail</code> methods. For example, <code>Submission.finishIssue(PolicyPeriod)</code> . Or, you can call <code>workflowInvokeTrigger(WorkflowTriggerKey triggerKey)</code> . Workflow actions during acknowledgment are optional. The workflow does not automatically move forward due to an acknowledgment.
	IssueSubmission	A message fired to notify an external system to issue a submission for a particular policy revision. The messaging plugin that submits the acknowledgment typically advances the workflow by finding the relevant workflow entities and calls their methods with names that begin with <code>finish</code> and <code>fail</code> . For example, call <code>Submission.finishIssue(PolicyPeriod)</code> . Or, you can call <code>workflowInvokeTrigger(WorkflowTriggerKey triggerKey)</code> . Workflow actions during acknowledgment are optional. The workflow does not automatically move forward due to an acknowledgment.
	PersonalDataPurge	A PolicyPeriod purge was committed as part of a personal data purge. The <code>EventFired</code> rule <code>PersonalDataPurge</code> , in the <code>EventMessage</code> rule set category, creates a message in response to the firing of the <code>PersonalDataPurge</code> event. You must define your own transport and destination for the message.
	PolicyPeriodAdded	Standard events for root entity <code>PolicyPeriod</code> .
	PolicyPeriodChanged	
	PolicyPeriodRemoved	
	PolicyPeriodPurged	The policy period was removed through quote purging.
	PreemptedPeriod	
	PremiumReport	A period needs a premium report in the billing system. This is a custom event that is part of the PolicyCenter integration with BillingCenter.
	ReinstatePeriod	A period needs to be reinstated in the billing system. This is a custom event that is part of the PolicyCenter integration with BillingCenter.
	RenewPeriod	A period needs to be renewed in the billing system. This is a custom event that is part of the PolicyCenter integration with BillingCenter.

Entity	Events	Description
	RewritePeriod	A period needs to be rewritten in the billing system. This is a custom event that is part of the PolicyCenter integration with BillingCenter.
	ScheduleFinalAudit	A final audit was scheduled, and PolicyCenter needs to notify the billing system. This is a custom event that is part of the PolicyCenter integration with BillingCenter.
	SendCancellationNotices	Tell an external system to send cancellation notices.
	SendNonRenewal	Tell an external system to send non-renewal data.
	SendNonRenewalDocuments	Tell an external system to send non-renewal documents. The messaging plugin that submits the acknowledgment typically advances the workflow by finding the relevant workflow entities and calls their methods with names that begin with <code>finish</code> and <code>fail</code> . For example, call <code>Submission.finishIssue(PolicyPeriod)</code> . Or, you can call <code>workflowinvokeTrigger(WorkflowTriggerKey triggerKey)</code> . Workflow actions during acknowledgment are optional. The workflow does not automatically move forward due to an acknowledgment.
	SendNotTaken	
	SendNotTakenDocuments	Tell an external system to send not-taken-related documents. The messaging plugin that submits the acknowledgment typically advances the workflow by finding the relevant workflow entities and calls their methods with names that begin with <code>finish</code> and <code>fail</code> . For example, call <code>Submission.finishIssue(PolicyPeriod)</code> . Or, you can call <code>workflowinvokeTrigger(WorkflowTriggerKey triggerKey)</code> . Workflow actions during acknowledgment are optional. The workflow does not automatically move forward due to an acknowledgment.
	SendRenewalDocuments	Tell an external system to send renewal-related documents. The messaging plugin that submits the acknowledgment typically advances the workflow by finding the relevant workflow entities and calls their methods with names that begin with <code>finish</code> and <code>fail</code> . For example, call <code>Submission.finishIssue(PolicyPeriod)</code> . Or, you can call <code>workflowinvokeTrigger(WorkflowTriggerKey triggerKey)</code> . Workflow actions during acknowledgment are optional. The workflow does not automatically move forward due to an acknowledgment.
	SendRescindNotices	Tell an external system to send rescind cancellation notices. The messaging plugin that submits the acknowledgment typically advances the workflow by finding the relevant workflow entities and calls their methods with names that begin with <code>finish</code> and <code>fail</code> . For example, call <code>Submission.finishIssue(PolicyPeriod)</code> . Or, invoke a workflow trigger with <code>workflowinvokeTrigger(WorkflowTriggerKey triggerKey)</code> . Workflow actions during acknowledgment are optional. The workflow does not automatically move forward due to an acknowledgment.
	WaiveFinalAudit	A final audit was waived, and PolicyCenter needs to notify the billing system. This is a custom event that is part of the PolicyCenter integration with BillingCenter.

Entity	Events	Description
PolicyPeriodWorkflow	PolicyPeriodWorkflowAdded PolicyPeriodWorkflowChanged PolicyPeriodWorkflowRemoved	Standard events for root entity PolicyPeriod.
PolicyTerm	PersonalDataPurge	A PolicyTerm purge was committed as part of a personal data purge. The EventFired rule PersonalDataPurge, in the EventMessage rule set category, creates a message in response to the firing of the PersonalDataPurge event. You must define your own transport and destination for the message.
ProducerCode	ProducerCodeAdded ProducerCodeChanged ProducerCodeRemoved	Standard events for root entity ProducerCode.
Reinstatement	ReinstatementAdded ReinstatementChanged ReinstatementRemoved	Standard events for root entity Reinstatement.
Renewal	RenewalAdded RenewalChanged RenewalRemoved	Standard events for root entity Renewal.
Rewrite	RewriteAdded RewriteChanged RewriteRemoved	Standard events for root entity Rewrite.
Submission	JobAdded JobChanged JobRemoved	Standard events for root entity Job.
ProcessHistory	ProcessHistoryAdded ProcessHistoryChanged ProcessHistoryRemoved	Some integrations can be done as a batch process. For example, use the event/messaging system to write records to a batch file (or rows to a database table) as each transaction processes. If all transactions process, submit the batch data to some downstream system. Write a batch process that starts manually or on a timer. To coordinate your software modules, use the ProcessHistory entity. If a batch process starts, a ProcessHistoryAdded event triggers. If you listen for the ProcessHistoryChanged event, check the processHistory.CompletionTime property for the datstamp in a datetime object.
SapCallHistory	SapCallHistoryAdded SapCallHistoryChanged SapCallHistoryRemoved	Standard events for SapCallHistory entities. The application creates one for each incoming web service call.
StartablePluginHistory	StartablePluginHistoryAdded StartablePluginHistoryChanged StartablePluginHistoryRemoved	Standard events for StartablePluginHistory entities. The application creates these to track when a startable plugin runs.
Administration events		
Group	GroupAdded GroupChanged GroupRemoved	Standard events for root entity Group.
GroupUser	GroupUserAdded GroupUserChanged GroupUserRemoved	Standard events for root entity GroupUser.

Entity	Events	Description
Role	RoleAdded RoleChanged RoleRemoved	Standard events for root entity Role.
User	UserAdded UserChanged UserRemoved	Standard events for root entity User. PolicyCenter triggers the UserChanged event only for changes made directly to the user entity, not for changes to roles or group memberships. A change to the user's contact record, for example a phone number, causes a ContactChanged event.
UserSettings	UserSettingsAdded UserSettingsChanged UserSettingsRemoved	Standard events for root entity UserSettings.
Contacts and address book events		
Adjudicator	ContactAdded ContactChanged ContactRemoved	Standard events for root entity Contact.
Company	ContactAdded ContactChanged ContactRemoved	Standard events for root entity Contact.
CompanyVendor	ContactAdded ContactChanged ContactRemoved	Standard events for root entity Contact.
Contact	ContactAdded ContactChanged ContactRemoved	Contact entities only exist as subtypes of Contact, such as Person. Those subtypes generate standard events for root entity Contact.
	PersonalDataPurge	A Contact purge was committed as part of a personal data purge. The EventFired rule PersonalDataPurge, in the EventMessage rule set category, creates a message in response to the firing of the PersonalDataPurge event. You must define your own transport and destination for the message.
ContactAutoSyncWorkItem	ContactAutoSyncWorkItemAdded ContactAutoSyncWorkItemChanged ContactAutoSyncWorkItemRemoved	Standard events for root entity ContactAutoSyncWorkItem.
LegalVenue	ContactAdded ContactChanged ContactRemoved	Standard events for root entity Contact.
PersonVendor	ContactAdded ContactChanged ContactRemoved	Standard events for root entity Contact.
PolicyContactRole	PolicyContactRoleAdded PolicyContactRoleChanged PolicyContactRoleRemoved	Standard events for root entity PolicyContactRole.
Place	ContactAdded ContactChanged ContactRemoved	Standard events for root entity Contact.
UserContact	ContactAdded ContactChanged	Standard events for root entity Contact.

Entity	Events	Description
	ContactRemoved	

Triggering a remove-related event

The *EntityNameRemoved* events trigger after either of the following occurs.

- Code deletes an entity from PolicyCenter
- 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 policy 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.

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 362

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 `ResyncAccount` event (triggered from an account 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 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.

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.

Generating new messages in Event Fired rules

Each time a system event triggers a messaging event, PolicyCenter 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 343

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 Policy// If the root object is a Policy...
```

Finally, at the third level there is a rule for handling each event of interest.

```
messageContext.EventName == "PolicyChanged"
```

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 policy = messageContext.Root as Policy
```

Once the Rule Engine recognizes the root object as a `Policy`, it allows you to access properties and methods on the policy to parameterize the payload of your message.

Next, create a message with a `String` payload.

```
var msg = messageContext.createMessage("The policy number is " + policy.PolicyNumber +
" and event name is " + messageContext.EventName)
```

Safe-ordering of messages

If you want to use the safe ordering feature of PolicyCenter, 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 policy " + policy.PublicID +
" and event name " + messageContext.EventName)
var msg2 = messageContext.createMessage("Message 2 for policy " + policy.PublicID +
" and event name " + messageContext.EventName)
```

You can also use loops or queries as needed. For example, suppose that if a policy-related event occurs, you want to send a message for the policy and then a message for each note on the policy. The rule might look like the following code statements.

```
var policy = messageContext.Root as Policy
var msg = messageContext.createMessage("message for policy with public ID " + policy.PublicID)

for (note in policy.Notes) {
    msg = messageContext.createMessage(note.Body)
}
```

This creates one message for the policy 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 cause value is " + policy.LossCause

if (policy.isFieldChanged("LossCause")) {
    msg = msg + " old value is " + (policy.getOriginalValue("LossCause") as LossCause).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. PolicyCenter 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 PolicyCenter 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 myPolicy = messageContext.Root as Policy;  
  
// generate the template and pass a parameter to the template  
var x = mycompany.templates.NotifyAdminTemplate.renderToString(myPolicy)  
  
// 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(myPolicyParameter : Policy) %>  
The Policy Number is <%= myPolicyParameter.PolicyNumber %>
```

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 PolicyCenter entities, such as `Policy` 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 PolicyCenter, there are multiple properties on `Message` for the primary entity.

Property	Description
<code>message.Account</code>	The account, if any, associated with this <code>Message</code> object.
<code>message.Contact</code>	The contact, if any, associated with this <code>Message</code> object.

Additionally, there is a `message.PolicyPeriod` property. However, `PolicyPeriod` is not a primary entity for PolicyCenter messaging.

If you set the `message.MessageRoot` property, the following behavior occurs automatically as a side-effect of setting this property from Gosu or Java.

Entity type	Description
Account or has an Account property	PolicyCenter automatically sets the <code>message.Account</code> property to set the primary entity.
Contact or has a Contact property	PolicyCenter automatically sets the <code>message.Contact</code> property to set the primary entity.
PolicyPeriod or has a PolicyPeriod property	PolicyCenter automatically sets the <code>message.PolicyPeriod</code> property. However, the <code>PolicyPeriod</code> property is not a property that affects the primary entity.

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, PolicyCenter can extract an Account from a `PolicyPeriod` entity, as well as other types of entities.

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. PolicyCenter uses that information to implement safe ordering of messages by primary object.

See also

- “[Sending safe-ordered messages](#)” on page 322

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 PolicyCenter, 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.

PolicyCenter also lets you save entities by name, saving references to objects with the message to update PolicyCenter 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.

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.

PolicyCenter 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 PolicyCenter 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. PolicyCenter 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.

Messaging interacts with PolicyCenter workflow

If you submit ACKS by using message plugins rather than SOAP APIs, you can make changes to data during the ACK. For example, you can update properties on entities. If you use workflows, your messaging code can advance a workflow. Search for the workflow entity and call its methods with names that begin with `finish` and `fail`. For example, call the `Submission.finishIssue` method. You also can call trigger invocation methods directly on the workflow objects, such as with `workflow.invokeTrigger`. Workflow actions during acknowledgement are optional, but be aware that workflows do not automatically progress forward simply from the acknowledgement.

The triggers that you can invoke are the ones that the workflow XML files define, so refer to the XML files for complete information. Remember to update messaging plugins if you change the workflow in ways that might affect invocations from messaging plugins or that call the `WorkflowAPI` web service APIs. For example, if you change a trigger's ID, delete a trigger, or remove a trigger, it may affect how the message responses use this workflow.

In some cases, messaging integration code directly interacts with workflow implementation. Be careful to coordinate changes in the workflow with changes for messaging plugins or other integration code that might rely on a specific implementation, such as a particular workflow trigger.

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, PolicyCenter 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

Validity and rule-based event filtering

PolicyCenter allows entry and creation of policies with fewer restrictions than might be true for an external system, such as a mainframe. PolicyCenter does this so that a new policy can be committed to the database with minimal information and the policy can be improved later as you gather more information.

However, in some cases, an external system might not want to know about the policy until a much more complete (or perhaps more correct) policy commits to the database. You might choose to not send a policy 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 policy. Each external system's destination defined in PolicyCenter may have completely different validation requirements.

For example, suppose an external system wants to be notified of every policy, but its standards were high about what kinds of policies it wants to know about. Additionally, you might want to omit notifying an external system about new notes on a policy that it does not know about yet. In other words, you did not send a message notifying the external system about the policy, 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 is one rule set that governs validation checking, called Policy Validation Rules. The validation rules engine sets the policy `ValidationLevel` property to the highest level that does not have rule rejection messages.
2. Your event rules check validation level (and other settings) – You can define Event Fired rules that define whether to send a message to an external system. For example, the rule might listen for the events `PolicyAdded`, `PolicyChanged`, 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 (policy.ValidationLevel > externalSystemABCLevel) {  
    // create message...  
}
```

Similarly, if you did not want to send events for a note before its associated policy is valid, you could write a rule condition such as the following.

```
note.policy.ValidationLevel > externalSystemABCLevel
```

It is important to understand that you define your own standards for event filtering and processing. PolicyCenter does not enforce any requirements about validation levels. The `EntitynameAdded` events and `EntitynameChanged` events trigger independent of the object's validation level.

Generally speaking, PolicyCenter 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 validation checking, called Policy Validation Rules. You can set validation levels in these rules that are checked later within the Event Fired rule set.

PolicyCenter itself does not actually use the validation level for any purpose.

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 `PolicyChanged` events, messages typically contain the latest information for the policy 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 policy to an external system. As part of the acknowledgment, the external system might send back its ID for the new policy. You can set the public ID in PolicyCenter to that external system's ID for the policy.

Suppose the next message separately sends information related to that policy to the external system. In this message, you want to include the new public ID for policy that was received from the external system in the acknowledgment. The external system can identify which policy belongs with this second message. If the public ID were merely set during the original processing of the event, the second message could not contain the new value from the external system. There would be no way to tell the external system which policy this second set of information goes with.

PolicyCenter 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 347

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 <AAAAAAA>.
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 `Policy`. 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 `Policy` 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 Policy
    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 357

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 <code>Messaging</code> editor, register the plugin name in the <code>Request Plugin</code> 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 <code>Messaging</code> editor, register it in the <code>Before Send Plugin</code> field, in which case the destination ignores the <code>Request Plugin</code> 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, PolicyCenter 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 PolicyCenter 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 PolicyCenter objects, such as `Policy`. 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:

[PolicyCenter/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.

PolicyCenter 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, PolicyCenter 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 PolicyCenter 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. PolicyCenter selects a message from the send queue.
2. If this destination defines a `MessageRequest` plugin, PolicyCenter 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, PolicyCenter 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. PolicyCenter 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, PolicyCenter 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, PolicyCenter 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>
Shut down 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 **PolicyCenterAdministration** tab. It is also possible for an external system, using web services provided by PolicyCenter, 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, PolicyCenter continues to use the original plugin instances.

IMPORTANT A plugin's suspend, resume, and shutdown methods must not call the PolicyCenter 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 PolicyCenter to the code XYZ for the external system.

If you implement your plugin in Java, you can use a utility class included in the PolicyCenter 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 n 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 MessageHistory	Can appear during re-sync
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	Legacy, non-retryable error. Provided for legacy use and no longer used.			

MessageStatus static property	Message status value	Meaning	Valid in Message	Valid in MessageHistory	Can appear during re-sync
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	
<code>isActive(state)</code>	Returns <code>true</code> if the state is in the array ACTIVE_STATES.
<code>isInFlight(state)</code>	Returns <code>true</code> if the status indicates a message in flight (PENDING_ACK).
<code>isRetryable(state)</code>	Returns <code>true</code> if the status is RETRYABLE_ERROR.
<code>isRetryable(state)</code>	Returns <code>true</code> if the status is in the array RETRYABLE_STATES.
<code>isPending(state)</code>	Returns <code>true</code> if the status is in the array PENDING_STATES.

MessageStatus static property or static method	Description
isPendingSend(state)	Returns true if the status is PENDING_SEND.
)	
isAcked(state)	Returns true if the status is in the array ACKED_STATES.
isError(state)	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 PolicyCenter 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, PolicyCenter 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, PolicyCenter 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 PolicyCenter message and an administrator resynchronizes the account.

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, PolicyCenter 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

PolicyCenter 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, PolicyCenter 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 send method, which triggers automatic retries potentially multiple times.	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. An administrator can retry the message from the Administration tab. Select the message and click Retry .

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>	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. In the base configuration of PolicyCenter, the typelist contains the following values. <ul style="list-style-type: none"> • <code>system_timeout</code> – timeout • <code>system_error</code> – general error communicating with external system • <code>contact_unsynced</code> – contact is not synced • <code>contact_error</code> – error connecting to contact manager You can call this method from the <code>MessageTransport</code> or when handling errors in your <code>MessageReply</code> plugin implementation.

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> <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("account", account)`. 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 367
- “Saving attributes of a message” on page 343
- “Custom events from SOAP acknowledgments” on page 337

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 368

Message error handling

PolicyCenter 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 an account, this message is safe-ordered. The messaging destination might be waiting for an acknowledgment on the previous message for that same account.
- 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. PolicyCenter can generate messages very quickly.

Errors during the send method

PolicyCenter attempted to send the message but the destination threw an exception.

- If the exception was retryable, PolicyCenter automatically attempts to send the message again some number of times before turning it into a failure.
- If the exception indicated a failure, PolicyCenter 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

PolicyCenter waits for an acknowledgment for a message it sent. If errors occur, such as the external system not receiving or properly acknowledging the message, PolicyCenter 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, PolicyCenter 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 PolicyCenter 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 PolicyCenter 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 PolicyCenter 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 PolicyCenter 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 PolicyCenter.

PolicyCenter 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.

PolicyCenter 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

PolicyCenter 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 PolicyCenter 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 PolicyCenter and corrects any related code, the external system might still have incorrect or incomplete data. PolicyCenter 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 PolicyCenter, views any unsent messages, selects a row, and clicks **Resync**.

The resync can be triggered from web services by using the `MessagingToolsAPI` web service method `resyncAccount`.

Note: Although `Contact` can be a primary entity for a PolicyCenter messaging destination, PolicyCenter does not support resynchronizing a contact.

As a result of a resync request, PolicyCenter triggers the resync event by calling `resyncAccount`. Configure your messaging destination to listen for this event. Then, implement Event Fired business rules that handle that event.

After a resync, PolicyCenter 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 PolicyCenter 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, PolicyCenter 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, "PolicyAdded".

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 358

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`).

PolicyCenter 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.

How resynchronization affects preupdate and validation

Preupdate and validation rule sets do not run solely because of a triggered event. The policy preupdate and validation rules run only if entity data changes. Triggered events that do not correspond to entity data changes do not cause policy 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 PolicyCenter, 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

PolicyCenter provides a simple user interface to view the event messaging status for policies. This helps administrators understand what is happening, and might give some insight to integration problems and the source of differences between PolicyCenter and a downstream system.

For example, if you add one policy in PolicyCenter but it does not appear in the external system, you need to know the following information.

- As far as PolicyCenter 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 policy 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 policy screen by selecting **Account Actions**→**Account Status** from the **Account** menu.

The PolicyCenter **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

PolicyCenter 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 *System 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 `WsAuthException`.
- 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 362
- “Messaging tools web service” on page 367

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 358
- “Messaging tools web service” on page 367

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 367
- “Message error handling” on page 363

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 367
- “Message error handling” on page 363

Resynchronizing an account for a destination

If PolicyCenter 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, PolicyCenter 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 an account, call the following method:

```
resyncAccount(destID : int, accountNumber : String)
```

The method can throw the following exceptions:

- If the account cannot be found, the method throws `BadIdentifierException`.
- If the destination is invalid, the method throws `IllegalArgumentException`.

See also

- “Resynchronizing messages for a primary object” on page 364
- “Messaging tools web service” on page 367

Purging completed messages

You can purge the completed messages stored in the `MessageHistory` table from the PolicyCenter database. Purging completed messages prevents the database from growing unnecessarily large with inactive messages. Additionally, you can reduce the time required to perform a PolicyCenter 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 367
- “Message error handling” on page 363

Suspending a destination

Suspending a destination typically means that PolicyCenter 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. PolicyCenter 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 315
- “Message error handling” on page 363

Resuming a destination

Resuming a destination generally means that PolicyCenter 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 315
- “Message error handling” on page 363

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 358
- “Messaging tools web service” on page 367

Getting the status of a destination

You can get information about the status of a message destination server.

Call the following method:

```
getDestinationStatus(destID : int)
```

The status is returned as a `String`.

Destination status is defined in the typelist `MessageDestinationStatus`. The possible values are:

- `Retrying`
- `Shutdown`
- `Started`
- `Suspended`
- `Suspending`
- `Resuming`
- `SuspendedInbound`
- `SuspendedOutbound`
- `Unknown`

This 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`.

See also

- “Message status code reference” on page 358
- “Messaging tools web service” on page 367

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 373

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, PolicyCenter can create multiple sender threads for each messaging destination to distribute the workload. These threads actually call the messaging plugins to send the messages. PolicyCenter

ignores this setting for non-safe-ordered messages because PolicyCenter 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 `WsIAuthenticationException`.
- If the destination ID is invalid, the method throws `IllegalArgumentException`.

See also

- “Message processing cycle” on page 320
- “Messaging tools web service” on page 367

Included messaging transports

Email message transport

The base configuration of PolicyCenter 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.JavaxEmailMessageTransport`.

See also

- *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, PolicyCenter provides a console message transport that writes the message text payload to the PolicyCenter 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.

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

Property	Setting
Initial Retry Interval	100
Retry Backoff Multiplier	2
Events	\w*

This configuration specifies that PolicyCenter is to send all events to this destination and trigger Event Fired rules accordingly.

7. Restart the PolicyCenter server.

Result

After restarting the server, watch the console window for messages.

Part 5

Policy-related integrations

Rating integration

Rating is the process of obtaining a price or set of prices for policy coverage. PolicyCenter supports rating from within PolicyCenter (internal rating) or an external rating engine. This topic discusses how to write a rating engine to work with the built-in PolicyCenter rating framework. It also describes the relationship between cost entity instances and cost data objects, the rating plugin, and how to implement a line-specific rating engine class.

The rating framework

PolicyCenter job workflows quantify the financial implications (the cost to the insured) from each job. Of course, submission jobs and policy change jobs have financial implications. In addition, other job types generate financial implications for the insured.

- Costs for renewals, such as renewal prices
- Costs for cancellations, such as refunds or cancellation charges

The entire process of offering a policy with a set of terms to an insured at a particular price is called quoting. Quoting includes the following steps.

1. Collect data about what the insured wants.
2. Validate the policy data.
3. Decide whether the policy requires underwriting approval.
4. Obtain costs or a set of costs for policy coverage for the insured. This step is called rating.
5. Finally, display the quote to the agent or customer.

This topic primarily discusses rating. Specifically, it discusses how to integrate your own rating code into PolicyCenter.

Many rating costs correspond to a specific object or coverage, for example a personal auto coverage. Costs can also represent taxes and surcharges that apply across an entire policy period. Costs can represent entire policies, in the case of umbrella coverage policies.

When PolicyCenter needs to rate a policy, it calls the registered implementation of the `IRatingPlugin` rating plugin interface to rate the policy. The rating plugin is the main entry point to the rating process.

To write your own code that rates policies, your main task is to implement the rating plugin, or modify the built-in implementation that PolicyCenter includes. The built-in implementation of the rating plugin is the Gosu class `gw.plugin.policyperiod.impl.SysTableRatingPlugin`. It uses other related classes such as `AbstractRatingEngine` and `CostData`. This documentation refers collectively to the rating plugin implementation its various related classes as the default rating engine. Review the built-in code to more deeply understand the default behavior of this rating engine.

If you use Guidewire Rating Management, you use the `PCRatingPlugin` rather than the `SysTableRatingPlugin`. The `PCRatingPlugin` is described later in this topic.

Any rating plugin implementation must perform two main tasks.

1. Alter the set of costs (cost entity instances) in the policy entity graph. The rating plugin has only one method. When the method returns, the branch must contain the correct set of cost objects that represent pricing for the entire policy period.

PolicyCenter later reads the cost information and creates onset and offset transactions. PolicyCenter sends the onset and offset transactions to your billing system.

2. Notify PolicyCenter that the full set of cost information (the quote) is complete and valid. If there are errors, the plugin marks it as invalid. The default rating engine code marks the quote valid by default. If the default rating code catches any exceptions, it logs the error and marks the quote invalid.

The default rating engine contains code to force a failed quote for certain test cases. If you use the built-in rating plugin, remove that code before pushing your code to a production system. Look in `SysTableRatingPlugin.gs` in the `ratePeriodImpl` method.

You might make only minor changes to the built-in rating plugin implementation. Most of the rating code for typical insurers are in separate rating engine classes for each line of business. Each line-specific rating engine extends the `AbstractRatingEngine` class. Additionally, consider encapsulating your actual algorithm into separate Gosu classes, one for each type of premium.

To modify rating for a built-in line of business, modify the built-in rating engine class or create a new subclass based on the built-in class.

To rate a new line of business, create a new subclass of `AbstractRatingEngine` that handles that line of business. If you create any new rating engine subclasses, you must also change the rating plugin method that instantiates and initializes the rating engine subclass based on the line of business. Do not forget to add that code.

The built-in rating engine general algorithm performs the following operations.

1. Find all change dates in the policy. First, find all dates on which anything changes in the policy in that branch. These are also known as slice dates.
2. Traverse down the graph on each slice date. For each date on which anything changes, traverse the policy graph downward from its root and calculate costs for this slice date. Each cost represents the cost of everything that can be rated in slice mode for that date. In PolicyCenter revisioning terminology, slice mode is a way of looking at a policy from a specific point in effective time. The abstract rating engine class automatically handles the date detection and revisioning logic associated with this step.
3. Calculate costs for each slice for every rating line. The abstract rating engine class next calculates the costs for the policy as it exists in effective time between the current slice date and the next slice date. However, costs generated in this step are not yet prorated. The rating engine works this way because it is a common design pattern to integrate with existing rating engines. Suppose an insured requests a policy change to be effective three days before the end of the policy period. The change adds five vehicles to the policy and it is the last change of the policy period in effective time. For this last slice date, the rating engine calculates the policy cost with new vehicles as if they were for the whole period, not just three days. During this step, each rating line sets some cost data properties, including effective and expiration dates based on the time between the current slice date and the next slice date. This is the step in which you might call out to an external rating engine. Be aware that the default rating engine assumes the rating engine responds synchronously.

This is the critical part of your rating code for costs that make sense in slice mode. Consider encapsulating your actual algorithm into separate Gosu classes, one for each type of premium.

4. Combine costs for all slices and prorate the costs. PolicyCenter merges any costs if two costs match all of the following conditions.
 - Costs are for the same type of premium, such as the same type of cost for the same car and same coverage
 - Costs are adjacent in effective time
 - Costs have the same rating result, in other words the same non-prorated premium and same rate

Next, the rating engine prorates slice mode costs based on the cost effective and expiration date properties set in the previous step. PolicyCenter skips costs that already have an actual amount in its `ActualAmount` property.

because that means that the rating engine already prorated it. The abstract rating engine class automatically handles date detection and revisioning logic associated with this step.

5. Calculate costs that depend on previous steps (window mode costs). Next, the line-specific rating engine rates the costs that apply to the entire policy period. For example, many types of discounts and taxes depend on getting sums of the slice-mode costs. If you support flat-rated costs, calculate them in this step.

This is the critical part of rating code for costs that make sense only as calculations using period-wide subtotals. Consider encapsulating your actual algorithm into separate Gosu classes, one for each type of premium.

6. Convert rating results into cost entity instances. The rating engine must adjust the rows in the database to match the rating results. Up until this step, all the rating happens on `CostData` objects. Cost data objects are non-entity class instances that mirror the role of actual `Cost` entity instances. There are subclasses of `CostData` for each cost type, effectively mirroring the various line-of-business-specific subtypes of the `Cost` entity. There are several parts of this conversion process.
 - a. Update reusable cost entity instances. If the rating engine can reuse an existing cost entity instance, the rating engine updates the cost to match an updated cost data object. The rating engine updates the cost's effective/expiration dates, all the rating-related properties, and all amount-related properties. The rating engine sets the properties in the cost entity instance to the values in the cost data object.
 - b. Create new cost entity instances if necessary. If the rating engine cannot find an existing cost entity instance for this cost, the rating engine creates a new cost. Generally speaking, this means that there is no cost that matches the cost's key values and is effective on the effective date of the cost data. The rating engine then adds a new cost by cloning an existing cost entity instance that matches on its same cost key but for different effective dates. This ensures that the new cost entity instance shares the same version list because they have the same fixed ID value.
 - c. Delete untouched cost entity instances. Cost entity instances from previous rating requests may no longer be relevant. For example, if you rate a policy and then remove a coverage, costs for that coverage no longer apply. If any cost entity instances do not match any cost data objects in the recent rating request, the default rating engine deletes the cost entity instances. Before rating begins, the rating engine makes a temporary list containing all cost entities for this policy. As the rating engine converts rating results into `Cost` entity instances, it removes any updated (reused) entities from the list. After the rating engine iterates across all cost data objects, any cost entity instances still in this list match no cost data objects in the current request. The relevant Gosu code refers to this as the untouched list of costs. The rating engine removes all entities in that list.

This general algorithm works for typical lines of business, and generally speaking results in easy-to-maintain and easy-to-understand code. It also helps that for slice mode changes, the abstract rating engine class handles the more complex parts of revisioning (date detection, cost merging, prorating) automatically.

Some lines of business are ill-suited to this approach, however. For example, in the United States for workers' compensation, regulators mandate a certain rating algorithm that varies greatly from the way most other lines of business generate costs.

Overview of cost data objects

PolicyCenter defines cost entities that describe the costs of different objects in the policy graph. These are the entities that persist in the database. A cost specifies how much money in the premium applies to which items in the policy. Cost objects share common properties. However, `Cost` is not a single database table. There is not a single `Cost` entity that is the supertype of each cost entity. Instead, the `Cost` entity is a delegate, which is similar to an interface definition. The `Cost` delegate defines the various properties common to all costs. Each line of business implements its own root entity table for its cost objects, and all cost objects for that line of business are subtypes of that entity.

For example, personal auto line of business defines its own `PACost` entity, and all personal auto costs are subtypes of `PACost`. The Businessowners Policy defines the `BOPCost` entity. Each line defines further subtypes of the line-specific cost entity, for example the personal vehicle coverage cost object entity `PersonalVehicleCovCost`. The line-specific cost subtype entities include additional properties and links into the policy graph specific to that line of business.

If you create custom lines of business, you might want to look at how the built-in lines of business structure their cost objects within the policy graph.

Although the cost and transaction entities define what PolicyCenter persists to the database to describe the results of rating, it is difficult to write code that directly modifies cost entities. There are subtle aspects to manage, such as splitting and merging entities, and this interacts with how PolicyCenter tracks revised entities across effective time. For example, creating or removing additional slice dates or tracking which changes are window mode changes.

To make it easier for customers to write correct and complete rating engines, the default rating architecture manages a parallel hierarchy of objects that correspond to each `Cost` entity. These objects are called cost data objects, implemented by subclasses of the `CostData` class. The cost data objects are Gosu objects, which are simply instances of in-memory Gosu classes. The cost data classes mirror all of the cost entity subtypes and contain the same basic information. Each cost data subclass mirrors each subtype of the `Cost` entity. In general, for every cost entity in the database there is one cost data object that the rating engine generates. The name of each cost data entity subclass is the name of its cost data object that it mirrors, followed by the suffix `Data`. For example, there is `PersonalVehicleCovCost` cost entity and so there is also a `PersonalVehicleCovCostData` cost data class.

Cost data objects are regular Gosu objects (not entity instances) that mirror every cost in the policy graph. During the first phase of rating, your rating engine creates cost data objects. During the second phase of rating, some code converts these cost data objects to `Cost` entity instances. If you use the built-in rating framework, this is easy and automatic.

In contrast to actual `Cost` entities, it is easy and low risk to write Gosu code that splits and merges `CostData` objects. You can optionally split or merge cost data objects in your integration code on the way out of PolicyCenter. You can split or merge cost data objects on the way in to PolicyCenter after external rating is complete. Afterward, your rating integration code can rely on built-in Guidewire code that intelligently merges cost data objects as appropriate. The built-in code updates `Cost` entities based on `CostData` objects that your rating engine generates.

This approach helps rating engines separate the two major processes of rating.

- Determining the appropriate rates and costs, including calling out to an external system if necessary
- Persisting rating information changes in the database in the most efficient and correct way possible

Guidewire strongly recommends that you write your rating engine to populate `CostData` objects instead of actual `Cost` entities. It is easier and safer to integrate rating using the built-in `CostData` objects and the built-in code that generates and manipulates them.

The following are notable differences between `Cost` entities and `CostData` objects.

Behavior	Costs (entity instances)	Cost data objects (Gosu objects)
Are they persisted in the database?	Yes	No. Cost data objects never directly persist in the database. Guidewire code defined in <code>AbstractRatingEngine</code> converts the cost data objects to costs which PolicyCenter persists in the database with the policy.
Are links between objects handled as direct references to entities?	Yes. The application entity layer handles database persistence just like all other entity data.	No. <code>CostData</code> objects do not use actual links to other entities. Instead, cost data objects refer to entities using the fixed ID values of the entity this cost represents. A fixed ID is a unique identifier that identifies an object across effective time and also across multiple revisions. Cost data objects do not directly store the fixed IDs. Instead, they encapsulate fixed IDs into Key objects, which are a container for foreign keys. This difference in link handling simplifies rating code, particularly with external rating engines. Send entity links as fixed IDs to an external system. With results from an external system, you can assemble cost data objects from the fixed ID information. The built-in default rating engine merges cost data objects as appropriate and updates the actual <code>Cost</code> entities. This means that you do not have to worry about as many details of the revisioning system.
Do PCF pages use them to display costs?	Yes. PolicyCenter PCF pages display cost entities attached to the policy graph.	No. After the rating engine completes, PolicyCenter discards all cost data objects, which are just temporary Gosu objects.

Behavior	Costs (entity instances)	Cost data objects (Gosu objects)
How easy is it to write your own code to split or merge costs across effective time?	It is difficult to split or merge Cost entities correctly. How the revisioning system deals with splitting and merging entities across effective time is complex. Fortunately, you do not need to manage this if you use the built-in code to manipulate cost data objects.	It is easy and low risk to write your rating code to primarily use cost data objects. Split or merge cost data objects in your integration code after your core rating engine code returns its results. Afterward, built-in PolicyCenter code updates the raw Cost entities based on your CostData objects.

Guidewire strongly recommends that you write your rating code to use the cost data architecture and the built-in code to manipulate them. This simplifies your rating integration code. In theory, your rating plugin can manipulate raw cost entities. However, this approach is challenging and not recommended for the reasons stated earlier.

A cost data class includes the following important properties and methods.

- Cost-related properties – Cost-related properties defined by the corresponding cost entity.
- Revisioning properties – Cost data objects duplicate revisioning-specific fields from the `Cost` entity, such as the effective date and expiration date.
- Methods that you override – Each cost data object class helps the rating engine with tasks such as finding the related cost entity instance for this cost data object.
- API methods that you can call – Every cost data class inherits built-in APIs from the `CostData` class.

Deciding appropriate granularity for your costs

A common question is what level of granularity is best to represent a cost in PolicyCenter.

Cost entity instances are the primary output of rating. Any non-premium data from rating that needs to be retained typically must be connected to a persisted cost entity instance.

When modeling your costs, consider the downstream usage of financial data, including general ledger, billing, agent commission payments and statistical reporting requirements. A general best practice is to store premium at the most granular level required and aggregate as needed later rather than split costs when sending to downstream systems. For example, create a `Cost` subtype and create multiple cost rows for similar data rather than requiring splitting a cost when sending to a billing system. Note that adding a typekey to the cost simplifies grouping, as done in PIP costs and Non-Owned liability costs in the Commercial Auto line.

However, creating very granular costs can create far too much data to store. In some cases, granular breakdown is only needed for audits. In that case, a single cost with additional data attached to that cost is better than splitting an object into many costs. An example of this multi-peril rating, where each peril requires a different algorithm that calculates a part of the premium. You can create one cost for the coverage because this is typically how it is charged and financially accounted.

Optionally, if you want to show the breakdown on the quote page or on reports, extend the `Cost` object with an array of objects containing details for each peril. The quote page and reports can optionally show this extra information.

Where to override the default rating engine

There are several different ways you can integrate your own rating engine into this flow, depending on what parts of the built-in system you choose to replace. First, it might help to understand the mechanics of the `AbstractRatingEngine` so you know where to intercede in that flow.

From a code-level perspective, the default rating engine has the following flow.

1. To rate a period, first PolicyCenter finds the implementation of the `IRatingPlugin` interface. PolicyCenter includes a built-in implementation that triggers the default rating engine behavior.
2. PolicyCenter calls the plugin implementation's `ratePeriod` method.
3. The rating plugin iterates across all policy lines. For each, the plugin creates a rating engine of the right rating engine subclass. All of these rating engine object are instances of the abstract rating engine class.

4. The rating plugin calls the `rate` method of each rating engine and passes the `PolicyLine` entity for the policy line as an argument.
 5. The built-in rating engines for standard rating lines (not worker's compensation) log some information and then call the `rateOnly` method to get the cost data objects.
 6. The `rateOnly` method has the following structure.
 - a. Get a list of slice dates for the policy. These are the dates that the policy changed in some way across effective time within the policy period.
 - b. With each slice date, slice the policy at that date. With each slice, call the `rateSlice` method to calculate cost data objects for each slice. If rating from the job effective date forward only, then get a list of cost data objects that represent all the costs from the database prior to that effective date. In other words, the rating engine had already calculated those slices, so there is no need to recalculate them. The `rateSlice` method is the typical place for implementing formulas for each type of premium, including rate table lookups.
- This is the critical part of your rating code for costs that make sense in slice mode. Consider encapsulating your actual algorithm into separate Gosu classes, one for each type of premium.
- c. Merge the cost data objects from all slices where possible.
 - d. Calculate prorated amounts. This step assumes linear proration. You can avoid proration for a cost by explicitly setting the actual amount (`CostData.ActualAmount`) property.
 - e. Call the rating engine's `rateWindow` method to get things like taxes and period-wide (non-linear) discounts.

This is the critical part of rating code for costs that make sense only as calculations using period-wide subtotals. Consider encapsulating your actual algorithm into separate Gosu classes, one for each type of premium.

7. After the `rateOnly` method returns, the `rate` method uses results to update the cost entities in the database.
8. The `rate` method validates the results.

There are three basic strategies you could use to integrate your rating integration code into PolicyCenter.

- Focus on rating each slice, and use the built-in architecture. This strategy is the easiest and safest approach.
- Use built-in cost data objects, but rewrite slice and window logic.
- Do everything on your own with raw cost entities. This strategy is not recommended.

The following table compares and contrasts these three strategies.

Strategy	What to override	Description
Focus on rating each slice, and use the built-in architecture	The rating engine <code>rateSlice</code> and <code>rateWindow</code> methods	<p>The easiest and safest approach is to let your rating engine conform to the PolicyCenter definition of slice costs.</p> <p>This strategy lets you leverage the built-in rating code within the <code>AbstractRatingEngine</code> class. In your rating engine, override one or both of the methods <code>rateSlice</code> and <code>rateWindow</code>. You probably need few if any changes to the built-in <code>rateOnly</code> method. PolicyCenter determines which slices in effective time to rate and how to merge and prorate costs. Run your internal rating algorithm or call out to an external rating engine from the <code>rateSlice</code> method. Your <code>rateWindow</code> method typically would be different logic, or a separate call to an external rating engine.</p> <p>Note that even if you handle slice costs in an external system, you might handle window mode costs such as taxes directly within PolicyCenter Gosu code. These types of costs are typically simpler to calculate.</p>
Use built-in cost data objects, but rewrite slice and window logic	The rating engine <code>rateOnly</code> method	If you do not want the built-in logic for handling slices, you can override the <code>rateOnly</code> method on each rating engine. For example, your <code>rateOnly</code> method might call out to your internal rating algorithm or an external rating system. Your <code>rateOnly</code> method must construct the full list of expected cost data objects. If you base your rating engine on the built-in <code>AbstractRatingEngine</code> class, your rating engine automatically handles generation and persistence of the <code>Cost</code> rows. However, if you are rewriting the <code>rateOnly</code> logic entirely, you must traverse the policy graph and prorate costs as appropriate.

Strategy	What to override	Description
		The worker's compensation line of business is a good example of when to use this approach. A slice-by-slice rating algorithm does not match regulator-imposed requirements.
Do everything on your own with raw cost entities	The entire rating plugin	This strategy is not recommended. However, you could choose to swap out the built-in rating plugin architecture completely and do something entirely different in your rating plugin. If you take this approach, you must manually calculate cost entities and attach them correctly to the policy graph. Although this offers you the most control over rating integration, it requires much more work to understand details of PolicyCenter revisioning to ensure you properly attach costs to the policy.

Common questions about the default rating engine

What are rate-scalable and basis-scalable costs?

A rating engine generates cost information that must include coverage-related charges, discounts, taxes, and fees. Certain operations work slightly differently when dealing with costs that are rate-scalable versus basis-scalable costs.

- Most insurable objects are rate-scalable. For rate-scalable objects, the rating looks up the rate based on a variety of factors such as the type of vehicle, the coverage terms chosen, the date and state. Often this calculation uses data from a table. Rating modifies that rate based on a variety of other factors (such as vehicle cost or driver age) to make a final adjusted rate. That final adjusted rate is generally the cost for the entire rated term. The final cost is that amount prorated over the portion of the policy for which the cost is appropriate.
- Other insurable objects are basis-scalable. A basis-scalable object means that the ratable basis for the object already considers time in some way. For example, it might capture the total payroll between two dates. Those costs are rated with a calculated adjusted rate multiplied by the basis in question, just like rate-scalable costs. However, PolicyCenter must not prorate basis-scalable costs based on time. This is because the basis amount already takes that time component into consideration. Notable basis-scalable objects include workers' compensation exposures and some general liability exposures.

Each cost data subclass indicates whether to merge that cost data subclass as rate-scalable or basis-scalable.

Can we always rate only from the current slice forward?

Some customers ask why they cannot always rate only from the date of change and later slices in effective time. The rating engine must know that if a policy change happens as of a given date, then no changes can happen that affect pricing prior to that date. For most lines of business, the PolicyCenter user interface enforces this requirement. However, worker's compensation and general liability policy changes allow changes that take effect at earlier effective dates. Workers' compensation rating is very different from standard lines, and heavily uses window mode costs. This is true about many fields in worker's compensation. Similarly, for general liability, these types of changes are possible for the exposure (location / class code) rows.

If backdated policy changes are possible, the only safe way to rate the period is to re-rate the entire period. In other words, if anything could have changed that affects the earlier part of the period (or the whole period), you cannot simply rate from the current date slice forward.

If backdated policy changes are impossible, it is safe to rate only from the current slice date. For normal in-sequence changes, this means just considering one slice date.

For out-of-sequence changes, PolicyCenter requires the rating engine to rate two or more slices.

Each line-specific rating engine can indicate that the current job can rate only from the current slice forward or whether it must re-rate the whole period. This flag is dynamic, and can you can override this calculation to use custom rules for existing lines of business or for new lines of business.

Choosing to rate from the change date forward is a performance optimization compared to rating the whole policy from the period start date. Even if you choose to use this optimization, PolicyCenter in some cases still rates the whole policy from the period start date, for example for a cancellation. Design your code to ensure that if

PolicyCenter rates the entire period or from the change date, the rating engine always returns the same consistent cost values.

See also

- *Application Guide*

Can we rate only if something important changed?

Some customers ask whether they need to re-rate everything or rate only the information that changed.

Deciding whether a change to something might theoretically impact a piece of premium associated with another object requires knowing all inter-dependencies in the rating algorithm.

For example, does the rating engine need to recalculate the premium for a vehicle-level coverage when any of the following events occurs?

- If the coverage changes?
- If the vehicle changes?
- If a vehicle modifier (such as an alarm system) changes?
- If the garage location changes?
- If some specific information about the driver changes? Some properties affect rating, some might not.
- If a multi-policy discount applies?

Many things that might at first glance seem distantly connected to the coverage might in fact cause the premium to change in real-world implementations.

Thus, it is much safer and easier to recalculate and check if anything changed. It is much better than trying to determine all possible changes that might theoretically impact a rate, and then checking only for those changes.

It is a painful maintenance challenge to ensure you record all potential dependencies as people continually change the rating algorithm over time. Mistakes due to lack of maintenance synchronization are extremely likely.

This is why PolicyCenter by default asks the rating engine to re-rate everything rather than selectively re-rating only changed data.

It is far safer to re-rate everything than try to maintain a list of dependencies that permit selective re-rating.

Can we rate the whole policy for each slice, rather than line by line

In some cases, rating a multi-line policy is required to be one slice at a time across all lines rather than rating each line independently. For example, one reason for this approach might be because the slice calculation includes discounts that depend on premiums across the entire policy. The default rating approach does not automatically support this requirement.

With additional customization, you can make this change by performing the following steps.

1. Create a new implementation of the rating plugin based on the default rating plugin, or customize the existing rating plugin. The built-in implementation of the rating plugin is the Gosu class `gw.plugin.policyperiod.impl.SysTableRatingPlugin`.
2. Modify the plugin to rearrange the logic of how to calculate slices and call out to the rating engines for each line. In the `ratePeriodImpl` method of the `SysTableRatingPlugin` class, find the following Gosu code.

```
for ( line in period.RepresentativePolicyLines ) {
    var ratingEngine = createRatingEngine(line)
    ratingEngine.rate()
}
```

3. Modify this code to instead iterate across slices instead of by rating engine (iterating across policy lines). The best thing is to review the code for the base class for rating engines, which is the class `gw.rating.AbstractRatingEngine`. Look at the method implementations for the method `rateOnly` and the private method `rateSlices`. You must copy that code or similar code to the rating plugin so that it iterates across each slices. For each slice, iterate across the rating engines for each line of business just like the code that you found in the previous step.

4. Additionally, make whatever special rating changes you require, such as the previously-mentioned example of discounts that depend on premiums across the entire policy.

This approach could allow you to get the advantages of using cost data objects and the `AbstractRatingEngine` helper methods, but implement looping the way you want.

Is proration always linear?

Each rating line determines the cost for each slice as if it were the cost for the entire term, even if it is only part of the term. For rate-scalable costs, which is the typical type of cost, the rating engine linearly prorates costs based on the percentage of the term that each cost is effective. In other words, PolicyCenter gets the term amount (`Cost.TermAmount`) then calculates the actual cost for this slice. This calculation uses the fraction of the days in the term this cost in this slice is active.

To get the length of time in the term, the default rating engine gets the `NumDaysInRatedTerm` property from the cost. The rating engine calculates the amount (`Cost.StandardAmount`) with the following pseudo-code that uses a fictional function `calcNumDays` that calculates the number of days between two dates.

```
var daysEffective = calcNumDays(Cost.ExpirationDate, Cost.EffectiveDate)
Cost.StandardAmount = Cost.TermAmount * (Cost.NumDaysInRatedTerm / daysEffective)
```

For most types of costs, this approach works for rate-scalable costs.

You might have a rare case in which you want non-linear proration. For example, a snow plow might have higher cost during winter months. You could represent the winter months as one type of coverage and the non-winter months with a different and lower coverage. However, if you want to represent the coverage as a single type of coverage cost but with non-linear proration, this requires changes to the proration plugin.

If you want to change this logic, refer to the class `gw.plugin.policyperiod.impl.ProrationPlugin` in the static inner class `ForGivenRoundingLevel` in the method called `prorateFromStart`.

There are two different places where the default rating engine assumes linear proration.

- In the proration step within the main merge-and-prorate part of the algorithm. You can workaround this by skipping proration by explicitly setting the standard amount (`CostData.StandardAmount`) and the actual amount (`CostData.ActualAmount`) properties on a cost data object.
- If you permit a line of business to rate only from the effective date forward, the default rating engine assumes linear proration in some of that code. The default rating engine constructs cost data objects for costs that are earlier in effective time than the job effective date. The default rating engine assumes that it can linearly prorate costs that span the job effective date.

How does proration handle minor differences in term length?

The default rating engine first calculates a term amount (`CostData.TermAmount`) and then prorates that value. To do this task, the code must know that the term is a certain amount of time.

To get the length of time in the term, the default rating engine gets the `NumDaysInRatedTerm` property from the cost.

In the default rating engine, the `NumDaysInRatedTerm` contains the number of days in a standard period from the period's effective date. Depending on the effective date, the number of days in a standard period can vary.

- A standard period of 6 months might be 182 days, 180 days, and so on.
- A standard period of 1 year might be 365 or 366 depending on the leap year.

This subtle difference has a couple implications that may not be obvious.

- The default rating engine charges the same rate even if the length of the standard period changes in a minor way. For example, 181 days compared to 184 days in a 6 month period. This has the advantage that to the insured, they understand that their rate did not change from period to period. However, some people do not expect this behavior.
- The default rating engine sets `TermAmount` based on a standard period length not the number of actual days in the term. Because of this, you can identify a non-standard period length even at submission or renewal time. For example, if the standard term is 365 days, a shortened period for 244 days will show 67% proration for costs that cover the entire period. The proration percentage is a percentage of the standard period length, not the actual period length.

You might want to take an entirely different approach to this proration algorithm. For example, suppose a rate from the table represents the price for 1 day or 30 days. You could perform the following operation.

1. Calculate the length of the actual policy period. For example, 280 days for a non-standard year period.
2. Determine the rate for that period with the following formula.

```
period_rate = actual_rate * (number_of_days_in_period) / (number_of_days_for_standard_rate)
```
3. Set the `TermAmount` property on the cost data object based on the 280 day rate.
4. Set the `NumDaysInRatedTerm` property on the cost data object to 280.

This way, the insured sees costs that represent the entire term that do not appear prorated. The rate varies if the period length varies, but this may be your desired behavior for some or all rating lines.

Can external rating engines handle proration directly?

The PolicyCenter default rating engine handles proration of costs. If you are using external rating engines to generate cost values, Guidewire recommends that you let the PolicyCenter default rating engine handle all proration of costs. To avoid subtle differences in handling proration and rounding errors, do not delegate proration to an external rating engine.

How are errors thrown by external rating engines handled?

External rating engines vary greatly in how they handle error conditions. Some systems return explicit error codes or throw exceptions. Other rating engines provide zero values in error conditions.

To avoid serious data integrity and business logic errors, research how your external rating engine handles error conditions in various cases.

As you parse data from your external rating engine, your code might need to check flags or check for zero premium to detect error conditions. Have conversations about error handling early in your implementation project. If your rating engine detects errors from your external rating engine, log the error and throw an exception from your Gosu code.

Optional asynchronous rating

Guidewire recommends you write your rating code to rate the policy synchronously. In other words, even if you need to call out to an external system, your code does not return to the caller until either it succeeds or it fails. By default, PolicyCenter assumes synchronous rating. The `QuoteProcess.requestQuote()` method calls to the rating plugin to rate the policy. After that completes, the `requestQuote` method then calls its own `handleQuoteResponse` method.

You can modify PolicyCenter to perform asynchronous rating if necessary. For example, instead of immediately returning a result and notifying the quote process, the rating engine generates a messaging event using the PolicyCenter messaging system. If you want asynchronous rating, you must make changes to the built-in code.

Enable asynchronous rating

Change PolicyCenter to use an asynchronous process for rating instead of the synchronous process that the base configuration provides.

Procedure

1. Find and edit the `gw.job.QuoteProcess` class in Studio.
2. Remove the call to `handleQuoteResponse`.
3. Decide where to inject your rating code. Carefully consider where you want to inject your rating code. The result has large consequences for how you design your asynchronous messaging code. For example, do you want just one message payload per policy, or one per slice date?

4. Wherever you decide to add your rating code, instead of a synchronous call to your rating engine, add the `RequestQuote` messaging event. A messaging event is a special signal that triggers the Event Message rule set in the same database transaction as the current Gosu code.
5. Write new Event Fired rule set rules that listen for the event and generate a message payload for your rating engine. It would contain important information from your policy so that it could generate prices for everything.
6. Write a messaging transport that calls out to your rating external system.
7. Eventually your messaging transport gets a response. During the message acknowledgement (message reply) part of your messaging plugin code, create and attach the raw cost entities based on the rating engine response.
8. Finally, your messaging transport tells PolicyCenter that rating is complete by calling the `handleQuoteResponse` method on the quote process.

Next steps

Whether you use synchronous or asynchronous rating, with the built-in rating engine architecture, if an exception occurs during rating, you must mark the quote as invalid.

See also

- “Where to override the default rating engine” on page 383
- “Triggering custom events” on page 337
- “Generating new messages in Event Fired rules” on page 338

Implementing rating for a new line of business

If you use the pattern of the built-in default rating engine, your main task is to modify the built-in rating engine classes or write your own one based on `AbstractRatingEngine`. The plugin contains the code that actually instantiates the rating engine objects. If you make your own rating engine subclasses, be sure to modify the rating plugin to create the correct rating engine subclass based on the line of business. This topic goes into further detail about what cost data objects and how to subclass `AbstractRatingEngine`.

The easiest approach is to use the built-in structure of `CostData` subclasses rather than directly producing `Cost` entities.

Your rating code can split, merge, prorate, and sum these `CostData` objects until your rating calculations finish. At that point, a built-in method of the `AbstractRatingEngine` class takes the list of `CostData` objects that each rating engine produces and compares them with what is in the database. This code alters the `Cost` entity instances in the database for this line of business. This code may do some combination of adding, modifying, splitting, and removing cost entity instances as appropriate. This is difficult code to write safely, and hence the benefit in abstracting the costs into the cost data objects.

Most built-in rating engine subclasses (although not all) implement rating using the following algorithm, implemented in part by the `AbstractRatingEngine` class.

1. Find all dates on which anything changes in the branch. In other words, find any entity that starts or ends in effective time. The `AbstractRatingEngine` class constructor implements this and sets the internal instance variable `_effectiveDates`. The method `rateSlice` uses this information variable’s contents.
2. At each of those dates, traverse down the tree (graph). As the code traverses down the tree, it produces costs for everything that can be rated in slice mode. Slice mode changes includes the common things like personal auto coverage and building costs. The resulting costs are not prorated and span from the slice date to the next slice date. The `AbstractRatingEngine` class method `rateSlice` implements this.
3. The rating engine rates each slice-mode cost. The `AbstractRatingEngine` class method `rateSlice` implements this. Your rating engine can override this method to rate each slice.
4. The rating engine merges back together any costs that are adjacent in effective time and which are equivalent, which means they have the same rates. There are methods in the base class `AbstractRatingEngine` that perform this task.

5. Your rating engine prorates costs that do not yet have an actual amount in the `ActualAmount` property.
6. Rate window-mode costs (like discounts and taxes) that might depend on sums of the prior costs.
7. Take the resulting set of cost data objects and adjust `Cost` rows in the database to match them.

This approach is used by all built-in rating engines other than the built-in workers' compensation rating engine.

You could describe this approach as rating down each slice and then merging costs back together. Think of this as the down-each-slice-and-merge approach. A theoretical alternative is to find all effective-dated split dates, or at least all relevant splits (this would be harder to determine). Next, rate across each object, such as a vehicle or coverage, calculating rates at each relevant point and then splitting off new `CostData` rows when the rates change. Such an approach avoids the need to remerge costs back together. However, it is more difficult to write easy-to-improve and easy-to-maintain code using this approach. As a result, PolicyCenter takes the down-each-slice-and-merge approach instead of the across-each-object-and-split-on-differences approach.

The logic for rating each particular type of object tends to share a common pattern. For rate-scalable types, the computation usually looks like the following operations.

1. Set the base rate to the result of some lookup. The lookup result might be based on coverage terms, date, state.
2. Set the adjusted rate to the base rate multiplied by various other factors. The other factors might include the underwriting company, driver ages, vehicle cost, and modifiers.
3. Set the term amount to the adjusted rate multiplied by the basis. Typically, the basis is 1.
4. Set the amount to the value of the following formula.

```
(the_term_amount) * (number_of_days_this_cost_spans / number_of_days_the_rates_are_for)
```

The amount computation happens during the proration phase mentioned above, not during the actual initial rating. For basis-scalable costs, the algorithm is similar. However, the basis is generally not 1 and the rating engine simply sets the amount to the term amount rather than prorated based on time.

See also

- For more information about rating integration strategy possibilities, see “Where to override the default rating engine” on page 383.
- For details of writing your rating algorithm, including why and how to use standard entities for table lookups, see “Coding your actual rating algorithm” on page 410.

What do cost data objects contain?

A cost data object (the `CostData` class) mimics a `Cost` entity, including its properties. A cost data object also includes the properties for prorating the amount and rate values. Just like for a regular `Cost` entity, properties for amount and rate values prorate according to the percentage of the term that the cost is effective.

For every line-specific `Cost` entity, each line of business must subclass `CostData` to create an equivalent cost data object for each `Cost` entity. The cost data entity has the same name as the entity followed by the suffix `Data`. For example, the personal auto line of business includes a cost entity called `PersonalAutoTaxCost`. This entity's corresponding cost data object class has the name `PersonalAutoTaxCostData`.

Each cost data class defines methods that tell PolicyCenter how to persist a `CostData` object persists to a `Cost` entity. As part of this, PolicyCenter needs to know how to combine similar cost data objects.

A cost data class includes the following important properties and methods.

- Cost-related properties – Cost-related properties defined by the corresponding cost entity. For example, the `PersonalAutoTaxCostData` class has equivalent properties for each cost-related property of the `PersonalAutoTaxCost` entity. It is important to note that any properties that represent links to other entity instances work differently in cost data objects compared to normal cost entity instances. Cost data objects use fixed ID Key objects to represent these links.
- Methods that you override – Each cost data object class must help the rating engine with tasks such as finding the related cost entity instance for this cost data object.
- API methods that you can call – Every cost data class inherits some built-in APIs from the `CostData` class. If you use the built-in default rating architecture (the Guidewire recommendation), the `AbstractRatingEngine` that

calls these. In that case, you do not typically need to call these APIs. If you do not use the built-in abstract rating engine or if you heavily modify it, you may need to use these APIs to implement rating.

See also

- “Cost data object methods and constructors to override” on page 396
- “Cost data APIs that you can call” on page 402

Cost core properties

The `Cost` entity is the core output of the rating engine. A `Cost` entity is a delegate not an actual entity. There is no one master database table for all costs. Each line of business defines its own root entity (each with its own database table) for its costs. For example, `PACost` and `BOPCost`. Each of these entities implements the `Cost` delegate. The line then defines further subtypes of that line-specific cost, such as `PersonalVehicleCovCost`, that include additional properties and links into the policy graph.

The `Cost` delegate defines the following various core properties that are common to all costs. For those properties, the cost information exists in multiple properties with different prefixes.

- If the property name has the prefix `Standard...`, this is the standard value for this cost information from the rating engine, before any overrides. For example, the `StandardBaseRate`.
- If the property name has the prefix `Actual...`, this is the actual value that PolicyCenter uses to calculate the premium. The actual rate is the value PolicyCenter sends to the billing system for this cost. For example, the `ActualBaseRate` property. This is a different value from the standard column only if overrides exist for that cost.
- If the property name has the prefix `Override...`, a user overrode the value with this new value. For example, the `OverrideBaseRate` property.

The following table includes the core cost properties. Except where noted, these are defined on the core cost delegate for costs entities (the data model delegate file `CostDelegate.eti`). Note that some properties have default values, whereas some are automatically calculated. Some properties you typically compute and explicitly set the values. See the `Description` column for details.

Cost information	Property names on costs and cost data objects	Description
Basis	<code>Basis</code>	The size of risk for the cost over the rated term. Usually this is directly from the object being rated, such as a workers' compensation policy or a general liability exposure. For owned property, typically the basis is one, such as one car or one building. For liability risks, the basis measures the amount of risk. For example, the amount of payroll for workers' compensation or the amount of sales for general liability. In your rating code, always explicitly set the <code>Basis</code> property.
Base rate	<code>ActualBaseRate</code> <code>StandardBaseRate</code> <code>OverrideBaseRate</code>	The cost rate percentage prior to applying any discounts or adjustments. The base rate typically comes from a rate table. In your rating code, set the property to one of the following settings. <ul style="list-style-type: none"><code>ActualBaseRate</code> – Always set explicitly. If there are overrides, the rating engine might set the actual property to zero to represent that it was unused.<code>OverrideBaseRate</code>, <code>StandardBaseRate</code> – Set explicitly only if the rating line supports overrides. If the user overrides a post-prorated cost, then the base rate may be irrelevant. Set the override property to <code>null</code> if there are no overrides.
Adjusted rate	<code>ActualAdjRate</code> <code>StandardAdjRate</code> <code>OverrideAdjRate</code>	The cost rate percentage after applying modifier factors such as discounts and adjustments. In your rating code, set the property to one of the following settings. <ul style="list-style-type: none"><code>ActualAdjRate</code> – Always set explicitly. If there are overrides, the rating engine might set the actual property to zero to represent that it was unused.<code>OverrideAdjRate</code>, <code>StandardAdjRate</code> – Set these only if the rating line supports overrides. If the user overrides a post-prorated cost, then the adjusted rate may be irrelevant. Set the override property to <code>null</code> if there are no overrides.

Cost information	Property names on costs and cost data objects	Description
Term amount	ActualTermAmount StandardTermAmount OverrideTermAmount	<p>The non-prorated premium amount or other cost for the entire term. This represents the amount if the cost were effective for the entire term. The term length is the value in the property NumDaysInRatedTerm. For the prorated value (using effective and expiration dates), see the row with label “amount”.</p> <p>In your rating code, set the property to one of the following settings.</p> <ul style="list-style-type: none"> • ActualTermAmount – Always set explicitly. • OverrideTermAmount – Set only if the rating line supports overrides. • StandardTermAmount – Set if the rating line supports overrides. Otherwise, setting it is optional. <p>If there are overrides, the rating engine might set any of these to zero to represent that it was unused. For example, if the post-prorated cost is overridden, the base rate may be irrelevant.</p>
Effective date	EffectiveDate	<p>The start date for the date range of this cost. In other words, this is the date that this cost becomes effective. This property is not part of the cost delegate definition, which defines most core cost properties. However, costs are revisioned entities, so this property automatically exists in every Cost entity.</p> <p>In your rating code, always explicitly set the EffectiveDate property.</p>
Proration method	ProrationMethod	<p>The proration method. The built-in choices are listed below.</p> <ul style="list-style-type: none"> • ProRataByDays – A prorated cost (the default) • Flat – A flat cost <p>With Guidewire Rating Management, you can define and rate flat costs. If you do not use Guidewire Rating Management, you can define flat costs, but you must configure rating for those flat costs. Define costs as flat costs by setting the ProrationMethod property to Flat on the cost data object.</p> <p>The system table rating plugin implementation contains built-in behavior to handle flat costs.</p> <ul style="list-style-type: none"> • In the CostData base class, the ProrationMethod property is copied between the cost data object and the cost entity instance. • In the CostData base class, the computeAmount protected method checks the value of the ProrationMethod property. If it is set to Flat, the method sets the Amount property to the term amount and does not prorate. <p>To add a custom proration method, perform the following operations.</p> <ol style="list-style-type: none"> 1. Add a new typekey to the ProrationMethod typelist. 2. In your CostData subclass, override the computeExtendedAmount method. Check the proration method and return the appropriate amount. <p>In your rating code, always explicitly set the ProrationMethod property.</p>
Expiration date	ExpirationDate	<p>The end date for the date range of this cost.</p> <p>In your rating code, always explicitly set the ExpirationDate property.</p>
Number days in rated term	NumDaysInRatedTerm	<p>The number of total days in a standard term, which PolicyCenter uses to determine the term amount. In other words, this value helps PolicyCenter prorate the term amount by comparing the number days in rated term to the effective and expiration dates. For example, 365 for a standard year term.</p> <p>In your rating code, always explicitly set the NumDaysInRatedTerm property.</p>
Amount	ActualAmount StandardAmount OverrideAmount	<p>The prorated amount of premium (or other cost) for the effective period, prorated to the effective date range defined by EffectiveDate and ExpirationDate properties. This is the term amount multiplied by the total effective days, divided by the number of days in the rated term (the property NumDaysInRatedTerm).</p> <p>In your rating code, set the property to one of the following settings.</p> <ul style="list-style-type: none"> • ActualAmount – It depends. The default code computes this during proration after cost merging. It is only set if it is not yet set. Always set explicitly for basis-scalable costs that are not prorated. Always set explicitly if cost's OverrideAmount was set,

Cost information	Property names on costs and cost data objects	Description
		<p>which means, the ActualAmount was overridden. An external rating engine might set ActualAmount explicitly to prevent merging and proration for some reason.</p> <ul style="list-style-type: none"> • OverrideAmount – Set only if the rating line supports overrides. • StandardAmount – It depends. If overrides are supported, then when initial rating completes, StandardAmount must have been set. If StandardTermAmount is set, the default code computes this property during proration after cost merging. The code only sets this property if it is not set. Always set explicitly for basis-scalable costs that are not prorated. An external rating engine might set StandardAmount explicitly to prevent merging and proration for some reason. <p>If the standard cost amounts (StandardAmount) are supposed to be prorated (derived from the term amount), then do not set the StandardAmount or ActualAmount properties. In the base configuration, the rating engine sets these properties. If null, the rating engine updates these properties in the updateAmountFields method in the <code>gw.rating.CostData</code> class.</p> <p>The ActualAmount property is very important. The property's value is what PolicyCenter actually charges the insured for this cost.</p>
Rate amount type	RateAmountType	<p>A typelist value that distinguishes between premium costs and non-premium costs. For example, this classifies a cost as one of the following types.</p> <ul style="list-style-type: none"> • Tax/surcharge • Standard premium • Non-standard premium <p>You can optionally set the RateAmountType property. The default value is StdPremium, which represents a standard premium, typically is correct. Only change this for costs on reporting policies.</p>
Subject to reporting	SubjectToReporting	<p>Will this cost be part of premium reporting for this policy if the policy supports premium reporting? If so, this value is true. Otherwise, false. The typical case for a policy is to not use premium reporting. If the policy does not use premium reporting, PolicyCenter ignores this property.</p> <p>Although much less common, some policies do use premium reporting, such as United States workers' compensation payroll reporting policies. For such cases, setting this property to true indicates that the given cost is not billed up front, in other words at the time of issuance. Instead, the cost might only contribute to the calculation of a required initial deposit. Later, when the premium reports complete, then PolicyCenter bills premiums for this cost. If the value of this property is false, then PolicyCenter bills this cost up front.</p> <p>You can optionally set the SubjectToReporting property. The default value false typically is correct. Only change this for costs on reporting policies.</p>
Overridable	Overridable	<p>Can this cost be overridden by editing this cost in the Premium Overrides screen? If this property is false, then PolicyCenter prevents users from editing it.</p> <p>Prevent overriding in the following cases.</p> <ul style="list-style-type: none"> • Costs that must never be overridden, such as altering a tax rate. • Costs for which the value is already configurable by a user somewhere else. For example, the schedule credit is already a discount cost. There is already a place in the application to edit the discount rate. You do not want a user to override that on the general purpose premium overrides page. Instead, it is best to edit that value in the proper more specific place in the user interface. <p>The default is true.</p> <p>Explicitly set the Overridable property if you do not want the default value (true). A rating engine can set the value on a specific cost if the line supports overrides but wants a particular cost to disallow overrides.</p>
Charge pattern	ChargePattern	<p>A division that determines how to combine (sum) the transactions that relate to this cost before sending it to billing. The charge pattern often correlates with the rate amount type (RateAmountType) property.</p>

Cost information	Property names on costs and cost data objects	Description
		In your rating code, always explicitly set the ChargePattern property.
Charge group	ChargeGroup	Another optional subcategorization for costs. In your rating code, always explicitly set the ChargeGroup property.
Override reason	OverrideReason	An explanation for why the user overrode this cost. The value is descriptive only. PolicyCenter does not use it to calculate premiums. Explicitly set the overrideReason property only if the rating line supports overrides.
Merge as basis scalable	MergeAsBasisScalable	Specifies whether during cost merging, whether to prorate the basis value based on the time length. Never explicitly set the MergeAsBasisScalable property. The property is read-only but non-final. Only override this property in CostData subtypes for basis-scalable costs. By default, it returns false.
Cost key	Key	An internal ID property that PolicyCenter creates on cost objects. Not directly mirrored on cost data objects. Never explicitly set the Key property. The property is read-only and final. You cannot override it or set it.
Intrinsic type	IntrinsicType	An internal ID property that PolicyCenter creates on cost objects. Not directly mirrored on cost data objects. Never explicitly set the IntrinsicType property. The property is read-only and final. You cannot override it or set it.

Cost overridable properties

The OverrideAdjRate, OverrideAmount, OverrideBaseRate, OverrideReason, and OverrideTermAmount properties are only relevant if the rating code for that line supports handling overrides. Otherwise, do not set them. Their values typically come from an existing Cost entity instance with user overrides. A cost data object has these properties to allow override information to move to a cost data object and back again to the Cost. Generally speaking, a rating engine does not set these override properties explicitly. Instead, call the `copyOverridesFromCost` method to set them.

The StandardAdjRate, StandardAmount, StandardBaseRate, and StandardTermAmount properties are relevant only if the line supports handling overrides. If so, the rating engine can optionally set them.

Effective and expiration dates

All the built-in rating engines set the effective and expiration dates explicitly. They do this in a centralized place and a consistent fashion.

When rating a slice, the built-in code sets the cost's effective date to the start of that slice. The code then sets the expiration date to the start of the next slice it will rate.

When at later time when adjacent costs merge, the code adjusts the effective and expirations dates accordingly.

If your core rating logic is in an external system rather than using our rating engine classes, remember to set the effective and expiration dates on new cost data objects. Pass the effective and expiration dates as constructor parameters to all CostData objects that you create. The process is demonstrated in the following code taken from `PASysTableRatingEngine`.

```

private function rateVehicleCoverage_impl(cov : PersonalVehicleCov,
    baseRate : BigDecimal, adjRate : BigDecimal) : PersonalVehicleCovCostData {
    var start = cov.SliceDate
    var end = getNextSliceDateAfter(start)
    var cost = new PersonalVehicleCovCostData(start, end, cov.Currency, RateCache, cov.FixedId)
    populateCostData(cost, baseRate, adjRate)
    return cost
}

protected function populateCostData(cost : CostData, baseRate : BigDecimal, adjRate : BigDecimal) {
    cost.NumDaysInRatedTerm = this.NumDaysInCoverageRatedTerm
}

```

```

cost.StandardBaseRate = baseRate
cost.StandardAdjRate = adjRate
cost.Basis = 1 // Assumes 1 vehicle year
cost.StandardTermAmount = adjRate.setScale(RoundingLevel, this.RoundingMode)
cost.copyStandardColumnsToActualColumns()
}

```

The `NumDaysInCoverageRatedTerm` is a method that you must implement in your rating engine. It is abstract in the superclass `AbstractRatingEngine`. It returns the number of days in a standard coverage term.

Adding line-specific cost properties and methods

Each line of business defines its root cost entity type. Every root `Cost` entity type must implement the `CostAdapter` interface and delegate it to a separate class that you create. This cost adapter interface has a single method called `createTransaction`. This method must create a `Transaction` object appropriate for that `Cost` type. Typically, the root `Cost` entity for a given line, such as `PACost` or `BOPCost`, implements this interface, so you do not need to reimplement it in any subtypes.

```

package gw.lob.ba.financials
uses gw.api.domain.financials.CostAdapter

@Export
class BACostAdapter implements CostAdapter {
    var _owner : BACost
    construct(owner : BACost) { _owner = owner }

    override function createTransaction( branch : PolicyPeriod ) : Transaction {
        var transaction = new BATransaction( branch, branch.PeriodStart, branch.PeriodEnd )
        transaction.BACost = _owner.Unsliced
        return transaction
    }
}

```

Most line-specific costs (subtypes of the root cost entity) must also implement a set of methods and properties specific to the line. The line encapsulates these required methods and properties in a line-specific interface with the name of the line followed by the suffix `CostMethods`.

For example, personal auto cost entities must implement all the methods and properties in the interface `PACostMethods`. Each subtype must override (implement) these methods.

The line-specific methods typically include methods that the user interface requires to display the cost or to collate entities for display in the Quote page.

For example, for the personal auto line of business, all costs must contain `Coverage` and `Vehicle` properties, which are links to the relevant coverage and vehicle entity instances.

```

package gw.lob.pa.financials

/**
 * Additional methods and properties provided by the costs that supply this interface.
 */
@Export
interface PACostMethods {
    property get Coverage() : Coverage
    property get Vehicle() : PersonalVehicle
}

```

These line-specific properties on a cost from these adapters also appear on the corresponding cost data object.

Fixed ID keys link a cost data object to another object

Each cost data object contains cost-related properties defined by its corresponding cost entity. For example, the `PersonalAutoTaxCostData` class has equivalent properties for each cost-related property of the `PersonalAutoTaxCost` entity.

A cost data object might contain the following types of properties.

- Properties in common to all costs in the line
- Properties unique to a specific cost entity

Generally speaking, cost properties that are in a cost object also exist also in the corresponding cost data class. For example, `PersonalAutoTaxCost` properties are on the `PersonalAutoTaxCostData` class.

There is an important difference between costs and cost data objects you must be aware of. In normal entities, links to entities typically take the form of real foreign key links in memory to other entities. For example, the property `PACost.Coverage` links to a coverage.

In contrast, `CostData` classes link to other objects using the fixed ID of any entity it references. The fixed ID is a unique identifier that identifies one object across effective time and also across multiple revisions.

In the database column itself, a fixed ID is just a numeric value. PolicyCenter encapsulates the fixed ID numeric value and the entity type for the object into a `Key` object. Think of the `Key` object as a simple container for foreign keys. In the context of rating, a fixed ID `Key` object is the most important usage of the `Key` class.

This difference in handling links simplifies rating code, particularly with external rating engines. You can easily send entity links as fixed IDs and the entity type to an external system. With results from the external system, you can assemble cost data objects from that information. Let the default rating engine merge related cost data objects and handle the complex revisioning code to create and update the actual `Cost` entities.

As you design your cost data object, add any line-specific properties that the equivalent cost object contains. Declare these as private properties whose names begin with an underscore. However, remember to declare properties that link to an entity instance as the data type `Key`. For example, the `PersonalVehicleCovCostData` class declares a link to the coverage as a private variable of type `Key`.

```
class PersonalVehicleCovCostData extends PACostData<PersonalVehicleCovCost> {
    var _covID : Key

    //...
}
```

To get the fixed ID `Key` object from an entity, simply access its `FixedID` property.

```
var fixedIDKey = myPersonalVehicleCov.FixedID
```

You can convert a fixed ID `Key` object to a standard direct link using the cost object method `setFieldValue`. Typical code uses this method to implement the cost data object method `setSpecificFieldsOnCost`, which copies line-specific (non-core) properties to the cost entity instance.

A critical difference between cost entity instances and cost data objects is how they treat links to other objects. Cost data objects store a fixed ID `Key`, which encapsulates a numeric fixed ID value and the entity type of the object. Other Gosu code in PolicyCenter use a `Key` object to track other types of foreign keys, but in rating-related code a `Key` typically contains a fixed ID.

Cost data object methods and constructors to override

To work properly with the built-in PolicyCenter default rating engine, any new subclass of `CostData` must override some methods and constructors. The following subtopics describe the methods that you must override.

Constructors for a cost data subclass

Cost data objects have two constructors that you must implement. For each of the two types of constructors already mentioned for the `CostData` class, there is a variant that you must use if you plan to support multicurrency.

If the rating engine determines that a coverage has a premium, it must create an entirely new cost data. This is the most common situation for needing a new cost data object. In this situation, PolicyCenter creates a cost data object with a constructor that takes effective and expiration dates.

To support this, all `CostData` subclasses require a constructor that takes the effective and expiration dates and optionally any other subtype-specific data (to set in private variables). The `CostData` subclass must pass the effective and expiration dates to its superclass constructor. The superclass constructor sets various defaults for cost properties.

Guidewire strongly recommends that the constructor includes any subtype-specific properties in the constructor rather than setting these properties after instantiation.

The following code was taken from the personal auto cost data subtype `PersonalAutoCovCostData`. This constructor takes the effective and expiration dates as parameters and calls the `init` method to set local variables.

```
construct(effDate : Date, expDate : Date, vehicleIDArg : Key, covIDArg : Key) {
    super(effDate, expDate)
    init(vehicleIDArg, covIDArg)
}

private function init(vehicleIDArg : Key, covIDArg : Key) {
    assertKeyType(vehicleIDArg, PersonalVehicle)
    assertKeyType(covIDArg, PersonalAutoCov)
    _vehicleID = vehicleIDArg
    _covID = covIDArg
}
```

If rating only from the date of change forward, then the rating engine must create cost data objects that represent existing costs prior to the change in effective time. The following steps occur.

1. The rating engine finds the relevant `Cost` entities that have an effective date prior to the effective date of the change.
2. For each cost data object, the rating engine creates a cost data object using the cost data constructor that takes an existing `Cost` entity. The constructor creates a cost data object that matches the existing cost.
3. If the `CostData` object has an expiration date that is later than the effective date of the change, the expiration date is set to the effective date of the change. The rating engine prorates the `CostData` object using the new expiration date.

For details, see the `AbstractRatingEngineBase` class in the `extractCostDatasFromExistingCosts` method.

To support this process, all `CostData` subclasses require a constructor that takes a `Cost` of the appropriate type. Contrast this with the other constructor. The other constructor takes the effective and expiration dates for a new initialized cost data but no existing `Cost` entity.

For example, if a rating engine's `extractCostDatasFromExistingCosts` method creates a `CostData` subclass, it calls this constructor. This method is only used when rating only from the date of change forward.

The following code was taken from the personal auto cost data subtype `PersonalAutoCovCostData`. This constructor takes the specific type of `Cost` entity as a parameter and sets local variables.

```
construct(c : PersonalAutoCovCost) {
    super(c)
    _vehicleID = c.PersonalVehicle.FixedID
    _covID = c.PersonalAutoCov.FixedID
}
```

For each of the two types of constructors mentioned for the `CostData` class, there is a variant that you must use if you plan to support multicurrency.

For the constructor that takes a `Cost` entity instance, the multicurrency variant takes an argument of type `PolicyPeriodFXRateCache`. The `PolicyPeriodFXRateCache` type is a cache of financial exchange rate data. This cache object is stored in every rating engine automatically when you based your rating engine subclass on `AbstractRatingEngine`. That base class populates a property called `RateCache`, which contains the `PolicyPeriodFXRateCache` object.

When your `CostData` subclass constructor calls its superconstructor, be sure to call the appropriate constructor and pass the multicurrency information. To support the rate cache from your rating engine, reference the `RateCache` property and pass it to the superclass.

For example, the cost data class for the business auto line (`BusinessVehicleCovCostData`) has a method signature that takes a `PolicyPeriodFXRateCache` object as a parameter.

```
construct(cost : BusinessVehicleCovCost, rateCache : PolicyPeriodFXRateCache) {
    super(cost, rateCache)

    // [ Do other things specific to this cost data subclass ]
```

From your rating engine, reference the cache in the `RateCache` property of your rating engine, as shown below.

```
override protected function createCostDataForCost(c : Cost) : CostData {
    switch (typeof c) {
```

```

        case BAStateCovVehiclePIPCost: return new BAStateCovVehiclePIPCostData(c, RateCache)
        case BAStateCovVehicleCost:   return new BAStateCovVehicleCostData(c, RateCache)
        case BAStateCovCost:         return new BAStateCovCostData(c, RateCache)
        case BusinessVehicleCovCost: return new BusinessVehicleCovCostData(c, RateCache)
        case BALineCovCost:         return new BALineCovCostData(c, RateCache)
        case BALineCovNonownedCost: return new BALineCovNonownedCostData(c, RateCache)
        default: throw "Unexpected cost type ${c.DisplayName}"
    }
}

```

Similarly, for the constructor that takes an effective date and an expiration date, the multicurrency variant takes two additional arguments.

- A currency (**Currency**) for the cost – The as-rated currency, which is the currency that was used for the rate calculation. This is different from *settlement currency*, which is the currency for billing the customer.
- A **PolicyPeriodFXRateCache** exchange rate cache object – The financial exchange rate cache for this rating engine.

```

construct(effDate : DateTime, expDate : DateTime, c : Currency, rateCache : PolicyPeriodFXRateCache) {
    super(effDate, expDate, c, rateCache)

    // Do initialization specific to your cost data subclass...
}

```

Key values for each cost data subclass

After the rating engine for each line of business generates cost data objects, the default rating engine merges cost data objects for similar costs. The rating engine considers two cost data objects mergable if they satisfy all of the following conditions.

- They have different but adjacent date ranges. For example, the date range January 1 through March 15 (end of day) and the date range March 16 through September 29.
- They have the same price, in other words the same value in the **TermAmount** property.
- They represent pricing for the same kind of cost for the same thing. For example, for a personal auto coverage cost data object, only merge the cost data objects if the objects share the same coverage ID and vehicle ID. This may require checking multiple properties, and even properties not directly on the object.

It is insufficient to check merely the fixed ID for the cost or the fixed ID of the target object. Technically, two costs can be equivalent and mergable even if they have different fixed IDs.

So, each cost data class must help the default rating determine whether two cost data objects are the same costs for the same things. The default rating engine asks the cost data class to provide a list of values that collectively identify which properties to compare for two cost data objects. Generally, that includes matching subtype-specific columns that uniquely identify the set of related costs and the object whose price this represents.

For example, as described in the bullet list earlier in this topic, suppose there are two personal auto coverage cost data objects. The default rating engine must merge the objects if and only if they have different date ranges and matching vehicle ID and coverage ID. This means that the vehicle ID and coverage ID values are what PolicyCenter calls the key values for the matching algorithm.

Do not confuse key values with **Key** objects that contain a numeric fixed ID value and an entity type. The fixed ID **Key** objects are how cost data objects store a link to a revised object within the policy graph.

Each cost data class must return a list of its own key values in its **KeyValues** property getter function. In other words, you must override the **property get** function for the **KeyValues** property. To continue the example of the personal auto coverage cost data, that cost data class returns a list that contains the vehicle ID and coverage ID from private variables. In Gosu this looks like the following code segment.

```

protected override property get KeyValues() : List<Object> {
    return {_vehicleID, _covID}
}

```

Setting this list properly is crucial for the PolicyCenter default rating engine to merge and convert cost data objects properly. Be extremely careful how you implement this method in new cost data classes.

Carefully consider how you implement the **KeyValues** method. Errors in this method corrupt rating data during the merge process.

PolicyCenter uses the key values that you return to create an instance of `CostDataKey`. That object encapsulates the key values that the default rating engine uses as it merges cost data objects.

How the rating engine uses the key values property

To help in the merge process, PolicyCenter has the concept of the cost key for a given cost. This topic provides some context about how the rating engine uses the key values property in the cost data object.

Do not confuse cost keys with `Key` objects, which simply contain a numeric fixed ID value and an entity type. The fixed ID `Key` objects are how cost data objects store a link to a revised object within the policy graph.

For `Cost` entity instances, the key has type `CostKey` and includes the following properties.

- The type of the `Cost`
- The `ChargePattern`, `ChargeGroup`, and `RateAmountType` columns
- Any other columns on the `Cost` that are not defined on the `Cost` delegate and not a standard policy graph property like `cost.EffectiveDate`.

Extension properties on the `Cost` delegate (in `Cost.etcx`) are not part of the `CostKey`.

In contrast, for cost data objects, the key has type `CostDataKey` and includes the following properties.

- The `CostData` subtype
- The `ChargePattern`, `ChargeGroup`, and `RateAmountType` properties
- Any values in the `CostData.KeyValues` property for that cost data subclass

Compare the following statements.

- Cost entity instances are the same kind of cost for the same thing only if the values match for all properties on their `CostKey` objects.
- Cost data objects are the same kind of cost for the same thing only if the values match for all properties on their `CostDataKey` objects.

Copying custom properties from cost data objects to entity instances

After a line of business creates its cost data objects, the default rating engine creates new cost entities or updates existing cost entities.

The root `CostData` class knows how to copy all the built-in cost properties from the cost data object to the cost object. However, for a cost data subclass, the default rating engine needs help to copy any subclass-specific properties to the corresponding new or changed cost entity.

To coordinate this with the default rating engine, each cost data subclass must override (implement) the method `setSpecificFieldsOnCost`. Before doing anything else, this method must call the superclass implementation of this method: `super.setSpecificFieldsOnCost`.

The cost data base class for each policy line does not need to call the superclass. There is no implementation of this method on the `CostData` root class. For personal auto, for example, the root class is the class `PACostData`. However, any subclasses of the cost data base class for each line first must call the superclass implementation.

Next, the `setSpecificFieldsOnCost` method must copy any additional properties from the cost data object subclass to the cost entity.

For non-foreign-key properties, simply set properties directly on the cost entity instance. Get values from private variables of the cost data object and set properties in the cost entity instance.

For foreign key properties, this approach does not work. Remember that for cost entities, links to other objects are regular foreign key links directly to the destination entity instance. In contrast, cost data objects store the entity's fixed ID encapsulated in a `Key` object. This difference simplifies rating code, particularly with external rating engines. You can easily send entity links as fixed IDs to an external system, and with results from the external system assemble cost data objects from that information. You do not have to worry about as many details of the revisioning system.

Because of this difference in links between cost entity objects and cost data objects, your `setSpecificFieldsOnCost` method cannot simply use code, such as the following code statement.

```
// This does NOT work because the cost data private variables like _line
// contain a Key object (which contains a fixed ID), not a real link to a PolicyLine entity
costEntity.PolicyLine = _line
```

To resolve this difference between how costs and cost data objects handle links, you can use a method called `setFieldValue` on the `Cost` entity. The `setFieldValue` method takes two arguments.

- The `String` property name on the `Cost` entity
 - A `Key` containing the fixed ID for the object to which you want to link. Alternatively, you can optionally pass a fixed ID `String` directly to this method instead of a `Key` if you have a plain fixed ID `String` for some reason.
- You might notice that some of the built-in rating code passes a fixed ID `Key` object to `setFieldValue`.

For example, the personal auto cost data subtype `PersonalAutoCovCostData` looks like the following code.

```
override function setSpecificFieldsOnCost(line : PersonalAutoLine, cost : PersonalAutoCovCost) : void {
    super.setSpecificFieldsOnCost(line, cost)
    cost.setFieldValue( "PersonalAutoCov", _covID )
    cost.setFieldValue( "PersonalVehicle", _vehicleID )
}
```

Getting versioned costs that match this cost data object

After you create cost data objects, the default rating engine must find the set of `Cost` entity instances that correspond to each set of cost data objects. In this context, a set refers to all cost data objects that share the same cost data object key values. The cost data object key values are a list of values that the property `get` function for the `KeyValues` returns.

The rating engine must find the corresponding costs entity instances for two different reasons.

- Find existing premium overrides to apply to a newly-created cost data object
- Use the cost data objects to update the `Cost` entity instances in the database

To find the set of versioned costs, the rating engine asks each cost data object. Every cost data class must override the `getVersionedCosts` method to provide this information. This is the most complicated function that a cost data subclass must implement. It is complicated because it requires careful application of knowledge about the PolicyCenter revisioning system.

The cost data object's `getVersionedCosts` method must return a list of version lists for the appropriate cost object type. A `VersionList` represents a set of cost rows that represent the cost for the same thing. Each row represents a different date range, and the price for that date range.

For example, for a `PersonalAutoCovCostData` object, the `getVersionedCosts` method returns a list of version lists that represent `PersonalAutoCovCost` entity instances.

The calling code expects to receive a `List` containing only one version list, or an empty list if no existing cost entities match this cost data object.

A general approach for the `getVersionedCosts` method is to first get a reference to the version list for the relevant object that has a cost. Use the `PolicyLine` method argument (which identifies the branch) and any private variables from your cost data object that contain `Key` objects.

Remember that a cost data object does not directly link to persisted entity instances. In the place of a direct link, the cost data object instead stores a fixed ID `Key` object. The fixed ID `Key` object encapsulates the entity type and the numeric fixed ID for the target object. The fixed ID uniquely represents the object within that branch (and across branches), across all of effective time.

Cost data objects store these fixed ID `Key` objects in private variables to refer to other entities. For example, a `PersonalAutoCovCostData` object links to a coverage with a fixed ID `Key` object for the coverage using its private variable called `_covID`.

To create a version list from a fixed ID `Key` object, use the `EffDatedUtil` static method `createVersionList`.

```
EffDatedUtil.createVersionList( branch : PolicyPeriod, fixedIDKey : Key)
```

The first argument is the root `PolicyPeriod` for that branch. You can easily get that from the `getVersionedCosts` first parameter of type `PolicyLine`. Simply get the `PolicyLine.Branch` property. The `getVersionedCosts` second parameter is the fixed ID `Key` that identifies the desired object with a cost. Get this information from a private variable in your cost data object. For example, the `PersonalAutoCovCostData._covID` property.

The `createVersionList` method returns that object's version list, which is an object that can provide information about a revised entity and how it changes across time. The compile-time type of this version list is an untyped version list (typed to the version list superclass). Cast it as appropriate to the more specific type to access type-specific properties on it.

For a real world example, a `PersonalVehicleCovCostData` gets the coverage version list as shown below.

```
// Get the version list for the auto coverage
var covVL = EffDatedUtil.createVersionList( line.Branch, _covid ) as PersonalVehicleCovVersionList
```

For the common simple case, the target object directly contains costs for only one thing and stores it the `Costs` property. In this simple case, the version list for this object also contains a `Costs` property. It is important to note that this version list property does not contain costs directly. Instead, it returns a list containing one version list that represents one or more database rows of `Cost` entities in the branch that represent that same thing. Your `getVersionedCosts` method can return this list containing a single version list.

The `Costs` property on the version list is an example of a generated property. Gosu dynamically adds it to the `PersonalVehicleCovVersionList` object because it is a property on a `PersonalVehicleCov` entity and contains an array of entities.

If no such cost object exists, the version list `Costs` property instead contains an empty list. In other words, it is a list containing zero version lists. Return that value to tell the rating engine that no such cost objects yet exist.

It is unsupported to return more than one version list in the list that you return from `getVersionedCosts`. In real world conditions, each cost data object corresponds to no more than one fixed ID, so no more than one version list is supported. The code that calls the `getVersionedCosts` method throws an exception if it gets more than one version list.

For example, the personal vehicle coverage cost data class (`PersonalVehicleCovCostData`) creates a version list and simply gets the version list property called `Costs`.

```
override function getVersionedCosts(line : PersonalAutoLine) :
    List<com.guidewire.commons.entity.effdate.EffDatedVersionList> {

    // Get the version list for the auto coverage
    var covVL = EffDatedUtil.createVersionList( line.Branch, _covid ) as PersonalVehicleCovVersionList

    // Get versionlist.Costs. It contains a list of 1 version list for costs, or is an empty list
    return covVL.Costs
}
```

However, some cases are more complex. In some cases the target object may have a `Costs` property but it represents multiple types of costs. For example, a personal auto coverage may support multiple vehicles. In such a case, there is a separate cost for each vehicle. This means that this method must iterate across all costs for the target object and determine which cost entity instances correspond to the current cost data object.

For example, for the personal vehicle coverage cost data class (`PersonalVehicleCovCostData`), some costs represent taxes, fees, or costs for vehicles other than the one we want. To find the right cost, use the `where` collection enhancement method and to determine if the cost matches the current cost data object.

In the personal vehicle coverage example, the Gosu block that tests each item must match a cost object if and only if all of the following are true.

- The cost is associated with the coverage for the cost data.
- The coverage's cost object has the right type (`PersonalAutoCovCost`).
- The coverage's cost object is for a vehicle whose fixed ID Key matches the vehicle ID stored in the cost data object.

This example uses a separate helper method called `isCostVersionListForVehicle`, which determines if the cost is for the same vehicle. Note that this helper method is actually comparing a version list for the cost not the cost entity itself.

```
override function getVersionedCosts(line : PersonalAutoLine) :
    List<com.guidewire.commons.entity.effdate.EffDatedVersionList> {
    var covVL = EffDatedUtil.createVersionList( line.Branch, _covid ) as PersonalVehicleCovVersionList
    return covVL.Costs.where(\ costVL -> isCostVersionListForVehicle(costVL)).toList()
}

[...]
```

```

private function isCostVersionListForVehicle(costVL : entity.windowed.PersonalAutoCovCostVersionList)
    : boolean {
    // Among all rows in the database for this cost, choose the one with the earliest effective date
    var firstVersion = costVL.AllVersions.first()
    return firstVersion typeis PersonalAutoCovCost and firstVersion.Vehicle.FixedId == _vehicleID
}

```

This example chooses the coverage version with earliest effective date. Although this choice appears arbitrary, cost cannot represent more than one car and a car cannot change its fixed ID value over time. Thus, you can get any coverage version from the version list, dereference its vehicle, and get the fixed ID from that entity instance.

Specifying whether to merge a cost as basis scalable

Each cost data subclass indicates whether to merge the `CostData` object as basis scalable.

To specify how to merge the costs, each cost data can implement a property getter function for the property `MergeAsBasisScalable`. In most cases, this property returns always `true` or always `false` for any class, but you can add custom logic if necessary to return `true` or `false` accordingly.

The rating engine behaves differently based on the value of the cost data property `MergeAsBasisScalable`.

- If `false`, the rating engine merges this `CostData` object using the standard method `mergeIfCostEqual`.
- If `true`, the rating engine instead merges this `CostData` object using the method `mergeAsBasisScalableIfCostEqual`. This method extends this cost data object to cover the period spanned by the other cost. Additionally, it adds together the `Basis`, `ActualTermAmount`, and `ActualAmount` properties if all the following things are true.
 - The passed-in `CostData` is effective as of the expiration date of this current cost.
 - The other cost is equal to this current cost as determined by the `isBasisScalableCostEqual` method.

The rating engine also uses the value of `MergeAsBasisScalable` to treat costs differently when creating `CostData` objects based on a set of existing costs and then prorating them.

The root class `CostData` always returns `false`. Cost data classes can override it as needed.

```

override property get MergeAsBasisScalable() : boolean {
    return true
}

```

An example of when you might use more complex logic in this property get function is general liability exposure costs. The built-in cost data subclasses returns `true` for costs with class codes that have an auditable basis, and `false` for any other class codes.

Cost data APIs that you can call

Every cost data class inherits some built-in APIs from the `CostData` class. If you use the built-in default rating architecture (the Guidewire recommendation), the `AbstractRatingEngine` that calls these. In that case, you might not need to call these APIs. If you do not use the built-in abstract rating engine or if you modify it, use these APIs as needed to implement your rating integration code.

You can call the cost data API methods, but you must not modify or override them.

Debugging cost data objects

For debugging use, it may be helpful to use the cost data object method `debugString`. This method generates a debug string containing the core rate, amount, and basis properties on the `CostData` for debug purposes.

APIs for merging costs

To merge two rate-scalable cost data objects together, use the `mergeIfCostEqual` method. If two cost data objects are adjacent in effective time and equal, then this method sets the expiration of the earlier cost to the expiration date of the later cost. The caller discards the later cost. Costs are equal if the `isCostEqual` method returns `true`. It checks that the two costs have the same basis, rates, and term amounts, along with a few other things.

To merge two basis-scalable cost data objects together, use the `mergeAsBasisScalableIfCostEqual` method. This method is similar to `mergeIfCostEqual`, but handles basis-scalable costs. However, this method is more complicated because merging basis-scalable costs requires adding together the basis and amount properties. Also, the criteria for equality are slightly looser, and defined by `isBasisScalableCostEqual`.

Getting cost entity instances for this cost data object

There are several related cost data APIs for getting (and optionally creating) corresponding cost entity instances for this cost data object.

To simply get a reference to the corresponding `Cost` entity instance, call the cost data object method `getExistingCost`. This method takes the `PolicyLine` to search. This method does not modify the effective window of the returned cost. It merely finds the matching cost object, if any, that is effective as of the effective date of this cost data object. If there is no existing cost, this method returns `null`.

Internally, this method calls the `getVersionedCosts` method, which each cost data must override.

```
var c = getExistingCost(myPolicyLine)
```

In some cases, you might want to actually create a new `Cost` row or modify an existing one. This choice depends on whether there is an existing `Cost` instance that corresponds to this cost data with the appropriate effective dates. The cost data object method `getPopulatedCost` implements this logic.

There are several possibly outcomes of `getOrCreateCost`.

- If there is an existing `Cost` at the effective date of this cost data object, this method returns it.
- If there is an existing `Cost` but none at the effective date of this cost data object, this method creates a new cost entity cloned from an existing cost. If there are multiple to choose from, the rating engine chooses the one with earliest effective date. Among other properties cloned, the fixed ID value for the new cost entity is always the same as cloned cost. This is important to remember. The fixed IDs must be the same because both costs represent the cost for a specific thing, just at different effective dates. The method returns the new cost object.
- If there is no existing `Cost`, even at other dates, the method creates a new cost entity. The new cost object has an entirely new auto-generated fixed ID. The method returns the new cost object.

Unlike the `getExistingCost` method, the `getOrCreateCost` method never returns `null`.

```
var c = getOrCreateCost(myPolicyLine)
```

Finally, there is another method called `getPopulatedCost`. It performs the following operations.

1. Calls the public method `getOrCreateCost` to get the matching cost object, or creates a new one. See the earlier description of this method in this section.
2. Calls an internal method `populateCostEntity` that populates the entity with the core cost properties from this cost data object.
3. Calls the public method `setSpecificFieldsOnCost`, which each cost data class implements to copy all the non-core (line-specific) cost properties from the cost data object to the `Cost` entity instance.

```
var c = getPopulatedCost(myPolicyLine)
```

Checklist for relationship changes in cost data objects

For new cost data subclasses, there are multiple things you must do carefully to ensure that your cost data works with the default rating engine. This topic summarizes changes for a cost data subclasses, either for new subclasses or modifications to add links to policy graph objects. This topic contains links to earlier topics that discuss some subjects in more detail.

The most important thing to add to a `CostData` subclass is an extra relationship (foreign key) to indicate where the cost links within the policy graph. For example, for a vehicle-related cost, the cost might attach to the graph at the level of a vehicle and line-level coverage. Within the cost data object, the foreign keys are fixed ID values encapsulated in a `Key` object.

To add another relationship (foreign key) to a cost data subclass, perform the following steps.

1. Add the foreign key as private variables on the `CostData` with the variable type `Key`. Prefix the name with an underscore, such as `_covid`.
2. Add new arguments to the constructor for these new values. In the constructor, set your private variables based on the constructor arguments.
3. Override the cost data method `setSpecificFieldsOnCost`. Set your foreign key properties when creating a new cost.
4. Override the cost data property `get` function for the `KeyValues` property. Add any new relationships to the list that this property get function returns. For example, for a vehicle-related cost, suppose there are two cost data objects that are otherwise the same but for different vehicles or a different coverage. By properly adding the key values for vehicle and coverage to the key values, the rating engine considers the costs different. The rating engine does not merge together or otherwise throw them away as duplicate.
If you customize built-in entities or create entirely new cost data subclasses, remember to add entity relationships to the `KeyValues` return value. Otherwise, new costs disappear during the merging process.
5. Override the cost data method `getVersionedCosts`. Use the new foreign key properties in the cost data object to find the existing costs (in the database) that are for the same thing. For example, for a line-level vehicle cost, this method must return only the ones that are for a specific vehicle and coverage.

If you customize built-in entities or create entirely new cost data subclasses, remember to consider your new entity relationships in your `getVersionedCosts` implementation. Otherwise, new costs disappear during the merging process.

The other typical thing a cost data subclass must do is to add extra properties that distinguish between costs that are different but that connect at the same level. For example, suppose you have several possible line-level taxes, such as state tax, uninsured coverage surcharge, and so on. You would need a tax type field that explains which tax this represents. Like a foreign key property, add this information as a new private variable in the class, add it to the constructor, and do all steps in the previous numbered list.

In some cases, you might need to add to cost data variables but not add them to the `KeyValues` return value or use them in the cost data method `getVersionedCosts`. For example, you might add the following types of information.

- Useful information about the cost data, such as its rating tier
- Information that does not uniquely identify the cost in the graph, for example the cost for state tax for vehicle 1 and coverage B

A common mistake is to add new entity relationships but not add them to the `KeyValues` return value or add within your `getVersionedCosts` implementation. This makes new costs disappear during the merging process. Be careful to do all of these steps correctly.

Writing your own line-specific rating engine subclass

The `AbstractRatingEngine` class is the parent class for all built-in line-specific rating engine classes. It serves two purposes.

- Defines a basic structure for rating that works for most lines
- Defines helper functions that could be useful across lines of business

The most important top-level method on `AbstractRatingEngine` is the `rate` method. The default rating plugin calls this method. This method is responsible for rating and then adjusting the `Cost` rows appropriately. Do not modify this method, generally speaking. It converts cost data objects to cost entity instances, adds them to the policy line, and removes no-longer-used costs on that policy line.

The `rate` method calls out to the `rateOnly` method. It rates costs per slice, then merges and prorates, and then finally rates window mode costs. That algorithm suffices for most lines of business. However, other lines such as workers' compensation and inland marine override the `rateOnly` method to work differently in built-in code. You could override `rateOnly` in a rating engine to perform your own custom behavior. The `rateOnly` method returns the list of `CostData` objects, which the `rate` method converts to `Cost` rows to persist in the database.

For typical rating lines, your actual formulas and table lookups would happen in the `rateSlice` and `rateWindow` methods. The `rateSlice` method must rate a `PolicyLine` viewed as a slice on a specified slice date. The `rateWindow` method must rate all the window-mode costs for the `PolicyLine`. If you write your own rating engine subclass, writing your actual rating formulas in these methods are your primary tasks.

For a typical rating integration project, implementing your actual rating algorithm, typically in `rateSlice` and `rateWindow`, is the majority of your engineering time.

In nearly all cases, it is best to use `AbstractRatingEngine` as the parent class for new rating engines. There is a parent class of `AbstractRatingEngine`, called `AbstractRatingEngineBase`. If the normal paradigm of rating one line of business at a time does not mesh well with the rating algorithm, the new base class allows this. For example, to integrate with an external rating engine that rates all lines of the package in a single call. Refer to `AbstractRatingEngineBase` in Studio for details.

Create a new policy line rating engine

You can change PolicyCenter to use a custom rating engine for a particular policy line.

Procedure

1. Create a new subclass of `AbstractRatingEngine` for the rating line. You must use Gosu generics syntax to parameterize the type with the line of business class. The example below is taken from the `CommercialPropertyLine` class.

```
class CPRatingEngine extends AbstractRatingEngine<CommercialPropertyLine> {
```

2. Configure your line of business to instantiate your new rating engine class. In your `PolicyLineMethods` class for your line of business, find the `createRatingEngine` method. This method must return an instance of your new rating engine class. Method arguments include a rate method (`RateMethod`) and a set of parameters as a `java.util.Map` object. Optionally use these arguments choose different rating engine subclasses. The following example instantiates a different rating engine if the rate method is `TC_SYSSTABLE`.

```
override function createRatingEngine(method : RateMethod,
    parameters : Map<RateEngineParameter, Object>) : AbstractRatingEngine<PersonalAutoLine> {
    if (RateMethod.TC_SYSSTABLE == method) {
        return new PASysTableRatingEngine(_line as PersonalAutoLine)
    }
    return new PARatingEngine(_line as PersonalAutoLine,
        parameters[RateEngineParameter.TC_RATEBOOKSTATUS] as RateBookStatus)
}
```

3. Create a `CostData` subclass for the line and further subclasses of `CostData` for each subclass of cost on the line.
4. Decide whether your line can be rated from the change date forward only or whether PolicyCenter must re-rate the whole period.
5. Override methods on `AbstractRatingEngine` to actually do the rate calculations. The exact methods might vary.
6. Consider encapsulating the logic for each type of rating in a separate class for each type of rating algorithm. You can put your actual rating formulas and table look-ups into these classes. Wherever your actual rating code is, separate your table look-up code from your formulas.
7. If this line handles premium overrides, add relevant logic.

Methods and properties of the abstract rating engine class

The following tables summarize the methods in the abstract rating engine class to override or use. The rightmost column indicates whether a new rating engine typically overrides this method.

The methods in this table are methods that a new rating engine would typically override.

AbstractRatingEngine method (or property as noted)	Description	Override it?
<code>createCostDataForCost</code>	Given the specified Cost entity instance, creates the appropriate CostData subclass and returns it. The <code>extractCostDatasFromExistingCosts</code> method calls this class. The implementation of the <code>createCostDataForCost</code> method must prepare for any Cost subtype that the <code>existingSliceModeCosts</code> method could return.	Yes, for typical rating lines. If you completely override the <code>rateOnly</code> method or you override <code>shouldRateThisSliceForward</code> to return <code>false</code> , this method is unused.
<code>existingSliceModeCosts</code>	When rating only from the change date forward, this method finds any existing slice-mode costs and returns cost data objects to represent them. Typically, you would omit some costs such as taxes, which PolicyCenter treats as costs that the rating creates in <i>window mode</i> not on a per-slice basis. This method must return costs currently on the period that correspond to costs that are generated during the <code>rateSlice</code> method.	Yes, for typical rating lines. If you completely override the <code>rateOnly</code> method or you override <code>shouldRateThisSliceForward</code> to return <code>false</code> , this method is unused.
<code>getStateTaxRate</code>	Returns the tax rate for the given state. The postal code can affect local sales taxes, but the built-in logic does not use it. The default rating engine returns varying numbers based on the state, but it is only a default implementation. Override this method to encode your own rates.	Yes
<code>NumDaysInCoverageRatedTerm</code> property	This property <code>get</code> function returns the number of days that in the standard term, not minor variations of it. This number is important so that your rating calculations can use the proper value for a table lookup. Your subclass of <code>AbstractRatingEngine</code> must implement this method. It is abstract in the superclass.	Required
<code>rateOnly</code>	The core rating loop, with the following actions. <ul style="list-style-type: none"> • Rate slices. If the rating line requests rating only from the change date forward, respect that request if possible. • Merge cost data objects • Prorate cost data objects • Rate window costs 	Only if you want to bypass the general logic of the rating engine
<code>rateSlice</code>	Rates a given slice of the policy for this rating line. The method has a policy line (<code>PolicyLine</code>) argument that already has its slice date set for this slice. The default logic calls this function once for every slice date in the policy.	Yes, for typical rating lines. If you completely override the <code>rateOnly</code> method, this method is unused.
<code>rateWindow</code>	Rates the policy in window mode. This is where you would create costs that meet either of the following conditions. <ul style="list-style-type: none"> • Depend on the sum of the previous slice costs • Span the entire period and you must rate them just once instead of once for each slice date <p>The argument is the version of the policy line that is earliest in effective time.</p>	Yes, for typical rating lines. If you completely override the <code>rateOnly</code> method, this method is unused.

AbstractRatingEngine method (or property as noted)	Description	Override it?
ShortRatePenaltyRate property	This property <code>get</code> function returns the rate to apply to the penalty for a short rate cancellation. Most built-in rating lines call this method. In contrast, workers' compensation does not use this because it uses a more complicated lookup based on the associated rating context (<code>WCRatingContext</code>). The default rating engine always returns 10% but you can override this.	Yes
<code>shouldRateThisSliceForward</code>	Determines whether to rate only from the effective date of the job forward. If this returns true, the rating engine tries to rate only from the current slice forward in effective time. Choosing to rate from the change date forward is a performance optimization compared to rating the whole policy from the period start date. Even if you choose to use this optimization, PolicyCenter in some cases still rates the whole policy from the period start date, for example for a cancellation. Design your code to ensure that if PolicyCenter rates the entire period or from the change date, the rating engine always returns the same consistent cost values.	Only if you want to change the default logic. See the topics mentioned earlier in this table row.

The methods in the following table are utility methods, helper methods for the default rating engine, or internal methods that you override only if you significantly change the logic.

Utility methods perform the following types of functions.

- Merge and prorate `CostData` objects
- Calculate the number of days in the rated term based on the default policy term for the product
- Calculate a demo tax rate
- Calculate a short rate penalty rate

AbstractRatingEngine method	Description	Override it?
<code>addCost</code>	Adds a cost data object to the internal list of cost data objects for this rating line. You probably do not need to use this API unless you greatly change the logic of the class.	No
<code>assertSliceMode</code>	Asserts that the specified revised entity instance is in slice mode. This is a general utility function you can use.	No. It is a general utility function you can use.
<code>assertWindowMode</code>	Asserts that the specified revised entity instance is in window mode.	No. It is a general utility function you can use.
<code>attachCostEntities</code>	Performs the following operations. <ul style="list-style-type: none"> • Converts the cost data objects to actual <code>Cost</code> entities • Removes any old <code>Cost</code> entities for this policy line that were untouched by the <code>rateOnly</code> method. 	No
<code>attemptToMerge</code>	If possible, merges two costs. It returns <code>true</code> if the merge succeeded, otherwise returns <code>false</code> . This method merges the costs either as basis scalable or not, as the costs specify in the <code>cost.MergeAsBasisScalable</code> property.	No

AbstractRatingEngine method	Description	Override it?
<code>extractCostDatasFromExistingCosts</code>	<p>This method take a list of costs and a cut-off date and performs the following actions.</p> <ul style="list-style-type: none"> • Extracts cost data objects from any existing slice-mode costs on the line. • Adds cost data objects to the internal list for any cost that is effective prior to the given cut-off date. • Prorates any costs that fall across the cut-off date boundary. In the case of normal non-basis-scalable costs, the method prorates by removing the actual amount (the <code>ActualAmount</code> property) so that the cost potentially can merge with another slice. After merging, the code re-prorates the cost after slice rating is complete. <p>The built-in rating engine only calls this if the method <code>shouldRateThisSliceForward</code> returns <code>true</code>.</p> <p>You probably do not need to use this API unless you greatly change the logic of the class.</p>	No
<code>getNextSliceDateAfter</code>	<p>Given a particular date, finds the next effective date following this date. The built-in rating logic uses this method when rating in slice mode to determine the next change date. This allows the rating logic to determine how many days long the current slice is.</p> <p>Use this method to set the effective and expiration dates of the cost.</p>	No
<code>mergeCosts</code>	<p>Merge any equal costs that are attributed to the same elements (they have matching cost keys) and adjacent in effective time. This method returns a new list of costs rather than modifying the existing list of costs in place.</p>	No
<code>prorateToCutOffDate</code>	<p>Takes a cost, a cut-off date, an amount, and a rounding level. This method prorates the specified amount from the effective date of the cost to the given cut-off date. If the amount is <code>null</code>, this method returns <code>null</code>. You probably do not need to use this API unless you greatly change the logic of the class.</p>	No
<code>rate</code>	<p>This is the entry point for the rating request. The rating plugin calls this method for each rating line. It performs the following operations.</p> <ul style="list-style-type: none"> • Calls the <code>rateOnly</code> method on this object, which generates cost data objects. • Calls the <code>attachCostEntities</code> method to convert the cost data objects to actual Cost entities. • As part of the <code>attachCostEntities</code> method, removes any old Cost entities for this policy line that were untouched by the <code>rateOnly</code> method. 	No. Typically do not override this. For typical rating lines, just override <code>rateSlice</code> and <code>rateWindow</code> . If you must bypass or rewrite the built-in rating control structure, generally speaking, override the <code>rateOnly</code> method, not the <code>rate</code> method.
<code>updateAmount</code>	<p>Prorates any costs that do not yet have an <code>ActualAmount</code> property set. To do the main task of this method, this method calls each cost's <code>updateAmountFields</code> method.</p>	No

AbstractRatingEngine method	Description	Override it?
validate	Ensures that for any given period of time, there is only one Cost entity instance with a given cost key (CostKey). If this method fails, it indicates that the rating logic failed. The rating logic, either produced duplicate costs or incorrectly defined the cost data model such that expected operations could produce duplicate costs.	No

Deciding whether to rate only from change date forward

Policy changes, reinstatements, and cancellations typically rate the policy only from the change date forward. The application user interface enforces the rule that users cannot make changes that affect prior dates or backdate rate changes.

PolicyCenter encapsulates this logic in the `shouldRateThisSliceForward` method on `AbstractRatingEngine`. If you want the rating engine to rate the line from the effective date of the job forward, return `true`. If you want to rate the whole period starting at the period start, return `false`.

Choosing to rate from the change date forward is a performance optimization compared to rating the whole policy from the period start date. Even if you choose to use this optimization, PolicyCenter in some cases still rates the whole policy from the period start date, for example for a cancellation. Design your code to ensure that if PolicyCenter rates the entire period or from the change date, the rating engine always returns the same consistent cost values.

The behavior of the built-in implementation is described below.

- Policy changes rate from the change date forward provided that the period start date did not change. The method returns `true`. If the period start date changed, the method returns `false`.
- Reinstatements rate from the change date forward provided that the period start date did not change. The method returns `true`. If the period start date changed, the method returns `false`.
- Cancellations always rate from the change date forward. The method returns `true`.
- All other jobs always rate the whole period. The method returns `false`.

Subclasses can override this method to return `true` if the rating engine must always re-rate the whole period for a particular line of business. If the line allows for edits that are prior to the effective date, typically it is necessary to re-rate the whole period. For example, in general liability or workers' compensation lines, any job can result in changes prior to the effective date of that job. This quality is a result of how users edit exposures in window mode (not a particular slice date). If a particular rating engine never rates from the change date forward, you can override the `shouldRateThisSliceForward`.

This is the case for the built-in general liability line of business rating engine because you can edit some things in window mode. Window mode edits can cause changes to the policy prior to the effective date of the policy change.

When rating from the change date forward, at the end of the rating, PolicyCenter must contain a complete set of `CostData` objects representing the entity graph and its costs. So, PolicyCenter must still create `CostData` objects for each `Cost` already in the database even if you do not intend to rate it.

To implement this, the rating engine for every line of business must implement the `existingSliceModeCosts` method. This method must determine which costs are slice mode costs (and thus might not be rated).

You also must implement the rating engine method `createCostDataForCost`. It must create a `CostData` object from an existing cost. The actual logic that operates using these methods is in the `extractCostDatasFromExistingCosts` method. It contains additional logic to un-prorate rate-scalable costs and to prorate basis-scalable costs or overridden costs.

Creating new cost data objects in your rating engine

The main task of your line rating engine is to create new cost data objects and set the rating-related properties.

For this task, wherever your rating code lives, use the `new` operator with the appropriate `CostData` subclass. Be sure to use the constructor with individual properties, not the constructor that takes an entire `Cost`. In other words, use the constructor that looks like the following code from the personal auto rating engine.

```
var cost = new PersonalAutoCovCostData(start, end, vehicle.FixedId, cov.FixedId)
```

Next, your rating engine must set all the rating-related properties on new cost data objects.

If PolicyCenter is configured to support multicurrency, you must call the alternate constructor that contains a financial exchange rate cache object.

```
var cd = new MyCostData(cost, RateCache)
```

In addition to creating an entirely new cost data objects from your main rating algorithm, a rating engine subclass must override the `createCostDataForCost` method. This method takes the cost as a parameter. The method must create and return a new instance of the appropriate subclass of `CostData`.

The built-in rating code only calls this method when rating from the slice date forward.

```
override function createCostDataForCost(c : Cost) : CostData {
    switch (typeof c) {
        case PersonalAutoCovCost: return new PersonalAutoCovCostData(c, RateCache)
        case PersonalVehicleCovCost: return new PersonalVehicleCovCostData(c, RateCache)
        case PersonalAutoPIPCovCost : return new PersonalAutoPIPCovCostData(c, RateCache)
        default : throw "Unknown cost type ${typeof c}.Name"
    }
}
```

Coding your actual rating algorithm

Consider encapsulating your actual algorithm into separate Gosu classes, one for each type of premium.

Wherever you put your actual rating algorithm, Guidewire strongly recommends that you cleanly and clearly separate your rating logic into two parts.

- Data from lookup tables, which typically insurer business users manage
- Programming logic in rating formulas, which typically IT departments own and manage

Split your rating logic so that most data comes from lookup tables. Business users can manage the lookup table data separate from the pure formulas in programming code.

It might be tempting to store table data in system tables, which the application throws away and rebuilds from configuration files on application startup. However, Guidewire strongly recommends that you store your table data as standard entity instances. In other words, store table data as entities that you define in the data model that persist in the PolicyCenter database. Create screens for your business users to directly edit this data. In real-world deployments, this approach is the most versatile for ensuring that business users with the right permissions can change rates without requesting help from the IT department.

If you store table data as standard entities, remember that you cannot use source code control to move rate tables from a development or test environment to a production environment. Your data is in the database itself not configuration files. You must create your own process for moving your tables as administrative data. You can export the data from the test server and then load it into your production server.

Guidewire strongly recommends that you store your table data as persisted entity instances in the PolicyCenter database. There are special requirements to consider for copying data to your production system with this approach. This approach lets business users manage the table data in the application user interface. This approach results in the most successful real-world PolicyCenter rating integration implementations.

The basic pattern for rating code is described below.

1. Define a table with effective dates, the underwriter company (`UWCompany`), the state, and so on.
2. Create a class that retrieves data from that table using the query builder APIs. The built-in rating engines extensively use these types of lookups.
3. Create screens for editing those rate tables in the web user interface (PCF files) so business users can manage the data.

For rate table lookups, it is important to know what date to use for any table lookup that uses a date. As you write rating code, pay extremely close attention to ensuring that you always use the appropriate date.

Remember also that PolicyCenter sometimes must recalculate rates even when nothing important to the rate actually changed. Because of this, you must guarantee that your formulas and rate tables always return the same amount if PolicyCenter re-runs the formula at a later date.

The following rules must be followed.

- Reference dates for lookups must be consistent. For example, always use the start of the period, not the current date or current time.
- Never backdate any rate or formula changes. If PolicyCenter needs to re-rate a rating request with the same reference date at a later time to a production system, your code must always return the same result. If you ever need to change the formula, you must preserve the old formula for any existing data. The formula change must take effect in the future, or at least after the reference date for all existing policies. To change a formula used in rating, your code would preserve the old formula for all your legacy policies but rates on newer effective dates can use the new formula. In other words, your code that actually generates the rate would perform the following operations.
 1. Check the effective date of the change
 2. If the change's effective date is before the formula's change date, your code must run your old formula. It is critical that you never permit backdated rates in production systems.
 3. If the change's effective date is after the formula's change date, your code must use your new formula.
- Avoid round-off errors. Be sure to avoid round-off errors in your proration code.

Of course, if you are using an external rating engine, be sure that your external engine follows this general pattern. If you have questions about supporting these rules, please contact Guidewire Customer Support.

You must carefully implement rates and formulas, and be particularly careful with changes with real production data. Reference dates must be consistent. Never backdate rate changes or formulas. Avoid round-off errors. If you fail to follow these rules, customer data may be affected, and PolicyCenter has undefined behavior.

Handling premium overrides

A line of business can optionally support premium overrides. Of the built-in lines, only the worker's compensation line supports premium overrides.

If a policy type supports premium overrides, then while calculating the premium for a particular cost data object, your rating calculation logic must determine the following situations.

- Is there an existing cost entity instance in the database?

To find any previous cost entity instance, call the `AbstractRatingEngine` object's method `getExistingCost`. Each line-specific rating engine must implement their own version of the `getExistingCost` method.

- If there is an existing cost entity instance, does it contain overrides?

◦ If there are no overrides for the properties that support overrides, use the standard properties. Standard properties are the pre-override values in properties with names that start with `Standard`. For example, `StandardAmount`. Because there are no overrides, your rating engine must copy the standard properties (the value before any overrides) to the actual properties. To accomplish this, the rating code for workers' compensation uses the cost data object method `copyStandardColumnsToActualColumns`.

Only use `copyStandardColumnsToActualColumns` for costs without overrides. Never call the method if the cost has overrides, because the method deletes any existing override data.

◦ If there are overrides, use the override properties. Override properties are properties with names that start with `Override`. For example, `OverrideAmount`. The rating engine needs to copy override properties to the actual properties.

Be aware that there is a cost data object method called `copyOverridesFromCost`. However, this is not intended for use during the actual rating calculation. The built-in code calls that method to save override values into a cost data object. This ensures that any old override values are not lost when converting to a cost data object and back to a cost entity at a later time. Some people call this process round-tripping those properties between the costs entity and cost data forms.

The built-in workers' compensation rating line shows how different override properties to adjust from using standard values to using overridden values.

Add support for premium overrides

You can change how PolicyCenter handles premiums for a policy line to support overriding the premium that the rating engine calculates.

Procedure

1. Check if there is an existing cost.
2. If so, check if there are non-null values for any override properties.
3. If there are overrides, copy the appropriate properties from the existing cost.
4. Find and use override properties in your rate table lookups and algorithms.
5. Optionally, disallow user overrides for a cost by setting the `Overridable` property on a new cost data object to `false`. The default value of `Overridable` is `true`.
6. Decide how you want to handle zero-cost costs. A real-world rating engine calculates that there is zero cost (\$0) for a particular cost in some cases. Typically, the rating engine does not generate a cost row for that zero cost. Because the premium overrides user interface allows you to override existing rows but not add new `Cost` rows, by default, you cannot override a zero-cost cost. To allow users to edit (override) this cost, you can choose to create a zero-cost `Cost` entity instance in these cases. If you want to do this, you must modify the quote screen to filter out all the zero-cost rows. To edit those rows, you must show these zero-cost costs in the premium override screen.

Rating premium reports

When PolicyCenter processes a premium report, it is essentially collecting data and calculating premiums for only a portion of the policy period. The premium report job has non-null values for its properties `AuditPeriodStartDate` and `AuditPeriodEndDate`. When rating the premium report, the rating logic calculates costs only for amounts overlapping with the audit period.

You can see examples of this in the built-in workers' compensation rating. The default rating code determines the overlap between each employee (`WCCoveredEmployee`) and the audit period. Next, it prorates the basis by the formula in the following pseudo-code.

```
(number of days of overlap)/(number of days in WCCoveredEmployee effective period)
```

Rating logic does not create costs whose effective dates fall outside the audit period. For example, suppose the policy period is from January 1, 2019 to January 1, 2020 and the audit period is for February 1, 2019 through March 1, 2019. Rating must not return any costs that are effective prior to February 1, 2019 or which expire later than March 1, 2019. To prevent mistakes, PolicyCenter detects this condition and throws an exception for any rule violations.

The cost and cost data property `SubjectToReporting` controls what happens to the rating and billing of certain types of costs for policies subject to reporting. The property is important if you implement rating for premium reports.

During premium reporting rating calculations, you might want to omit some items from calculations, or handle them differently during rating. For example, the following behaviors are employed when handling the workers' compensation rating line.

- The expense constant is meaningful only for the period as a whole, not for any single monthly or quarterly report.
- Taxes are billed up front and at final audit. However, premium reports omit them.
- Rating can determine the premium discount percentage only by looking at the total premium for the entire period. PolicyCenter estimates this at submission or renewal time and determines a discount percentage. During premium reports, the system uses the percentage determined previously. It does not calculate a new discount percentage. This is because the premium report does not calculate a full period premium so it would have no way of deciding which discount percentage is appropriate. At final audit, PolicyCenter calculates the correct final discount and uses it to determine final audited premium.

A rating line example for personal auto - PersonalAutoCovCostData

This topic describes the implementation of one of the built-in lines of business, Personal Auto (PA). Although all lines of business are different, you can use this information to understand the types of things that you need to implement for rating integration for new policy lines. This implementation relies on the built-in architecture for rating typical lines of business.

This topic describes the default rating engine for Personal Auto, which uses system tables. The class name is `PASysTableRatingEngine`. If you use Guidewire Rating Management, you use a different built-in plugin implementation class than the default rating plugin.

For the personal auto line, the `Cost` entity hierarchy reflects how the line calculates various subtotals. The following table lists the `Cost` entity subtypes for this line, with a description and the name of its cost data object.

Personal auto cost entity	Description	Personal auto cost data class
<code>PACost</code>	The root entity type for all personal auto costs. This includes a link to <code>PersonalAutoLine</code> and an array of <code>PATransaction</code> entities.	<code>PACostData</code>
<code>PAMultiPolicyDiscCost</code>	A cost value that represents multipolicy discounts	<code>PAMultiPolicyDiscCostData</code>
<code>PAShortRatePenaltyCost</code>	A cost value that represents a cancellation short rate penalty.	<code>PAShortRatePenaltyCostData</code>
<code>PersonalAutoCovCost</code>	A cost value that adds a link to a <code>PersonalVehicleCov</code> and a link to a <code>PersonalVehicle</code> . This line uses this cost to join together a line-level coverage and a particular vehicle, since line-level coverages are priced for each vehicle.	<code>PersonalAutoCovCostData</code>
<code>PersonalAutoTaxCost</code>	A cost value for taxes	<code>PersonalAutoTaxCostData</code>
<code>PersonalVehicleCovCost</code>	A cost value for a vehicle coverage that has a link to <code>PersonalVehicleCov</code> . The link to the coverage already implies a link to the vehicle.	<code>PersonalVehicleCovCostData</code>

The `CostData` classes directly mimic the `Cost` hierarchy for this line. There is one root `PACostData` class. It contains a link to the `PersonalAutoLine` (a Key that contains the value of its fixed ID). There are subclasses that correspond to each of the `Cost` subclasses, each with its own properties (also stored as fixed ID Key objects) where appropriate.

A close look at PersonalAutoCovCostData

Look at the methods on the `gw.lob.pa.rating.PersonalAutoCovCostData` class, which is the most complex of the personal auto cost data objects. This class extends the `PACostData` class, parameterized on the corresponding entity type `PersonalAutoCovCost`. The class declares its superclass in Gosu generics syntax as shown below.

```
class PersonalAutoCovCostData extends PACostData<PersonalAutoCovCost> {
```

The angle bracket notation indicates the use of Gosu generics. This declaration shows that the class extends the `PACostData` generic class to work with the class `PersonalAutoCovCost`. This syntax allows the definition of `PACostData` to be written in a general way to work with many types of data, but be type-safe at compile time. Some of the rating code uses this parameter to determine which `Cost` entity subtype to create.

The class contains properties for the vehicle's `FixedId` property and the coverage's `FixedId` property. These fixed IDs are encapsulated in a `Key` object.

```
var _vehicleID : Key
var _covID : Key
```

Constructors for PersonalAutoCovCostData

The standard constructor initializes the class properties by calling the superclass constructor and the private `init` method. The `init` method ensures that the fixed ID Key objects passed to the constructor have the proper type. This check avoids errors such as switching the order of the arguments.

```
construct(effDate : Date, expDate : Date, vehicleIDArg : Key, covIDArg : Key) {
    super(effDate, expDate)
    init(vehicleIDArg, covIDArg)
}

private function init(vehicleIDArg : Key, covIDArg : Key) {
    assertKeyType(vehicleIDArg, PersonalVehicle)
    assertKeyType(covIDArg, PersonalAutoCov)
    _vehicleID = vehicleIDArg
    _covID = covIDArg
}
```

The class has an alternative constructor that populates a cost data object based on an existing `Cost` row. The superclass constructor does most of the work. This version just needs to extract out subtype-specific properties, in this case the vehicle and coverage fixed ID Key objects.

```
construct(c : PersonalAutoCovCost) {
    super(c)
    _vehicleID = c.PersonalVehicle.FixedId
    _covID = c.PersonalAutoCov.FixedId
}
```

Setting specific properties on cost for PersonalAutoCovCostData

The cost data `setSpecificFieldsOnCost` method needs to populate the line-specific (non-core) properties.

The `setSpecificFieldsOnCost` method first calls out to the superclass version, then sets the coverage and vehicle IDs. This code calls the `setFieldValue` method using the fixed ID Key object. Foreign keys on effective-dated (revisionable) entities store the value of the `FixedId` of the referenced entity. The alternative is to try to find an actual reference to the entity in question by traversing all over the entity graph, which would be resource-intensive and error-prone. Thus, calling `setFieldValue` is the easiest approach.

In the policy auto line, the `setSpecificFieldsOnCost` method is shown below.

```
override function setSpecificFieldsOnCost(line : PersonalAutoLine,
    cost : PersonalAutoCovCost) : void {
    super.setSpecificFieldsOnCost(line, cost)
    cost.setFieldValue( "PersonalAutoCov", _covID )
    cost.setFieldValue( "PersonalVehicle", _vehicleID )
}
```

Versioned costs for PersonalAutoCovCostData

Probably the most complicated method to implement on `CostData` objects is the `getVersionedCosts` method. It finds the `VersionList`, if any, of the existing cost on the coverage that points to this vehicle. Since it has only the fixed ID of the coverage, rather than an actual `PersonalAutoCov` object, it cannot call `_cov.VersionList`.

However, this code can construct a `VersionList` object based on the branch and the fixed ID of the Coverage.

To do this, use the `EffDatedUtil.createVersionList` method. The resulting version list is typed to the root type of all version lists. Thus, the code casts it as the more specific type, `PersonalAutoCovVersionList`. Next, the code gets the `List` of cost version lists, and finds the one (if any) that corresponds to the vehicle in question.

```
override function getVersionedCosts(line : PersonalAutoLine) :
    List<gw.pl.persistence.core.effdate.EffDatedVersionList> {
    var covVL = EffDatedUtil.createVersionList( line.Branch, _covID ) as PersonalAutoCovVersionList
    return covVL.Costs.where(\ costVL -> isCostVersionListForVehicle(costVL)).toList()
}
```

In this case, the method calls a private method called `isCostVersionListForVehicle` to do most of the work. To determine if a given `PersonalAutoCovCostVersionList` applies to this vehicle, it looks at the first (chronological) version and check the `FixedId` of its associated `Vehicle`.

```
private function isCostVersionListForVehicle(
    costVL : entity.windowed.PersonalAutoCovCostVersionList) : boolean {
    var firstVersion = costVL.AllVersions.first()
    return firstVersion typeis PersonalAutoCovCost and firstVersion.Vehicle.FixedId == _vehicleID
}
```

Key values for PersonalAutoCovCostData

The key values for this cost are just the vehicle and coverage ID.

```
protected override property get KeyValues() : List<Object> {
    return {_vehicleID, _covid}
}
```

Rating slice details for PersonalAutoCovCostData

The `PASysTableRatingEngine` class implements the `rateSlice` method by using the following general algorithm. The structure is similar to other lines of business.

- If the slice is canceled, do nothing.
- Otherwise, loop across the line-level coverages. Within that loop, loop over each vehicle and rate the combination of that line-level coverage and the vehicle.
- Loop over each vehicle. Within that loop, loop over each coverage on the vehicle and rate the coverage.

Since the argument passed to the `rateSlice` method is already sliced as of the appropriate effective date, the slice rating code is simple. It does not need complex code to determine effective dates. It can just traverse the graph in the normal straightforward way.

Look at the standard collision coverage to see more details of rating a coverage. The `rateSlice` method calls to `rateVehicleCoverage` for each coverage on a vehicle.

```
private function rateVehicleCoverage(cov : PersonalVehicleCov) : PersonalVehicleCovCostData {
    assertSliceMode(cov)
    switch (typeof cov) {
        case PACollisionCov: return ratePACollisionCov(cov)
        case PAComprehensiveCov: return ratePAComprehensiveCov(cov)
        case PARentalCov: return ratePARentalCov(cov)
        case PATowingLaborCov: return ratePATowingLaborCov(cov)
        default:
            PCFinancialsLogger.logDebug("Not rating ${typeof cov}")
            return null
    }
}
```

The `switch` statement using the type of the coverage is a fairly standard pattern in the built-in rating code, within the `rateSlice` method of a line-specific rating engine.

Let us now look at the rating code for the personal auto collision coverage.

```
private function ratePACollisionCov(cov : PACollisionCov) : PersonalVehicleCovCostData {
    var ratingDate = ReferenceDatePlugin.getCoverageReferenceDate(
        cov.Pattern as CoveragePattern, cov.PersonalVehicle)
    var baseRate = getRateFactor("papCollRate", "base", ratingDate)
    var adjRate = baseRate
        * getRateFactor("papCollDeductible", cov.PACollDeductibleTerm.OptionValue.OptionCode, ratingDate)
        * getAgeFactor(cov.PersonalVehicle, ratingDate)
        * getRateFactorInRange("papVehicleCostNew", cov.PersonalVehicle.CostNew as double, cov, ratingDate)
        * getDriverRatingFactor(cov.PersonalVehicle, ratingDate)
        * getNoLossDiscountFactor(cov.PersonalVehicle.PALine, ratingDate)
        * getUWCompanyRateFactor(cov.PersonalVehicle.PALine)
    return rateVehicleCoverageImpl(cov, baseRate, adjRate)
}
```

The structure in this example is typical. First, figure out the date on which to look up rates. Next, look up a base rate factor in the demonstration system tables used for personal auto rating. Multiply those factors by various modifiers, such as the following items.

- Chosen deductible
- Vehicle age
- Vehicle cost
- Diver's record
- Loss history
- Underwriting company

In this example, the last line of the method calls out to the common `rateVehicleCoverage_impl` function. That function sets the effective date span of the cost to be one of the following dates.

- Start date = the start of the slice to rate
- End date = the start of the next slice date

The implementation is shown below.

```
private function rateVehicleCoverage_impl(cov : PersonalVehicleCov, baseRate : BigDecimal,
                                         adjRate : BigDecimal) : PersonalVehicleCovCostData {
    var start = cov.SliceDate
    var end = getNextSliceDateAfter(start)
    var cost = new PersonalVehicleCovCostData(start, end, cov. Currency, RateCache, cov.FixedId)
    populateCostData(cost, baseRate, adjRate)
    return cost
}
```

The method calls a standard `populateCostData` method that is common to both vehicle-level and line-level coverage costs. The method copies the rates into the cost, sets the term amount, and copies the standard columns into the actual columns.

```
private function populateCostData(cost : CostData, baseRate : BigDecimal, adjRate : BigDecimal) {
    cost.NumDaysInRatedTerm = this.NumDaysInCoverageRatedTerm
    cost.StandardBaseRate = baseRate
    cost.StandardAdjRate = adjRate
    cost.Basis = 1 // Assumes 1 vehicle year
    cost.StandardTermAmount = adjRate.setScale(RoundingLevel, this.RoundingMode)
    cost.copyStandardColumnsToActualColumns()
}
```

The `NumDaysInCoverageRatedTerm` method is abstract in the superclass `AbstractRatingEngine`, so you must implement this method in your rating engine. The method returns the number of days in a standard coverage term.

The result of the `rateVehicleCoverage` call is a fully-populated `PersonalVehicleCovCostData` object. Other code can now add this object to the list of `CostData` objects.

The `AbstractRatingEngine` class might later merge this `CostData` with other adjacent `CostData` objects. Next, the `AbstractRatingEngine` prorates the cost. Lastly, the `AbstractRatingEngine` built-in methods convert the cost data object to a cost entity instance and persist that object to the database.

Rate window method for PersonalAutoCovCostData

The `rateWindow` method in `PASysTableRatingEngine` rates the following items.

- Multipolicy discounts
- Cancellation penalties
- Taxes

The multipolicy discount is notable because it rates essentially in slice mode, however just using the effective dates of the modifier itself. For each effective span, the discount sums the costs between that point in time and applies the discount to that period of time.

The cancellation penalty simply applies a flat percentage to the sum of all the costs computed up to that point in time (including the discount costs). Finally, tax calculations apply a percentage to the sum of all costs (including the cancellation penalty).

Rating variations

Rating for some lines of business may vary significantly from the approach exemplified by the personal auto line. The following sections discuss notable differences for very different built-in lines of business. Even if you do not need to modify rating for these lines of business, these documentation topics might help orient yourself to potential rating variations.

Workers' compensation rating

Rating in workers' compensation is different from other lines of business. One reason is that the algorithm used varies by state. Another is that users need the ability to evaluate and adjust the premium basis, payroll, of the entire policy term even when a change is initiated midterm. PolicyCenter accomplishes this in the user interface by displaying the entire policy term information even when affecting a midterm change (window mode). When a rating element or factor changes midterm, you may the adjust payroll details into separate rating periods within the window of the policy term.

Sometimes an insurance company must rate the period as a split period, which means a requirement to rate an annual policy split into two or more periods. For example, a policy must be rated midterm to give the insured new rates mandated by the state. Each of these reasons to split the period causes the following operations.

- PolicyCenter creates a new `RatingPeriodSplitDate` object, sometimes called an RPSD.
- PolicyCenter makes the `CoveredEmployee` exposures (class code, basis amount) split on that date.
- Modifiers also sometimes split, depending on a setting for the modifier pattern in the product model.

Fundamentally, worker's compensation rating performs the following operations.

1. Calculate manual premium – Manual premium means applying the standard rates as published in *the manual* to each covered employee. This is standard for all states. The rating engine iterates across the exposure rows in the database. PolicyCenter applies the following calculation.

```
premium = calc_basis * rate * factor
```

The code statement contains the following elements.

- `calc_basis` is usually the payroll amount
- `factor` is the multiplication factor. Typically this is $1/100$ since rates typically represent every \$100 of payroll.

This number is non-prorated (does not use effective time) because PolicyCenter already adjusted the basis for the length of time represented by the row. This is the meaning of the PolicyCenter term basis scalable costs. PolicyCenter scales the basis itself, and in this case does not use any time-based proration or scaling factor.

2. Calculate additional premium amounts by state and rating period – The rating engine iterates across each state and rating period. For each, the rating engine calculates additional premiums for increased limits factor, experience modifier, and many other things. These steps vary by state, and frequently change over time.
3. Calculate premiums that apply to the entire period – Final pass calculations that apply to the entire period might include premium discounts based on the total annual premium. An annual premium must apply to the policy as a whole, not for any given rating period.

The implementation for the built-in workers' compensation rating algorithm is the class `gw.lob.wc.rating.WCRatingEngine`. Look at the code for the `rateOnly` method to see the fundamental split between the manual premium and the remaining non-manual-premium calculations.

1. Manual premium calculations – The manual premium costs create `WCCovEmpCostData` objects linked to exposures (the `WCCoveredEmployee` objects).
2. Additional premiums calculations – The other calculations create `WCJurisdictionCostData` objects at the state level. The by-rating period logic is clear within this step.
 - a. First, the engine iterates across all combinations of state and rating period.
 - b. For each one of these combinations, the rating engine gets the subtotal of all manual premiums within that state and rating period.

Note that `WCCoveredEmployee` objects can never span more than one rating period. PolicyCenter enforces this at the application level. If necessary, PolicyCenter splits the object, one for each period. This means that as the rating engine iterates by state and rating period, the premium for each `WCCoveredEmployee` (the `WCCovEmpCostData`) falls into exactly one combination.

The manual premium rating for each exposure is simple. The rating engine uses a system table (`rates_workers_comp.xml`) to look up the class code of an exposure. The table returns a rate and minimum premium for each class code by effective dates, state, and so on. The rating engine tracks the largest minimum premium for each state. The rating engine later uses the largest one as the required minimum premium.

There is also some extra logic to handle rating overrides. The rating engine calculates the amount ignoring overrides. It uses a previous override if one exists.

Refer to the Gosu class `gw.lob.wc.rating.WCCoveredEmployeeRater` for more details.

Tables driving rating cost algorithm definition

The rating engine calculates the rest of the costs with an algorithm driven by tables. The basic approach is to read the set of steps required by state and date from a system table (`WC_Rating_Steps.xml`). The steps have a strict order.

The user interface needs to know to display certain price subtotals after certain steps. There are a couple of times where you need to get to a certain stage of the calculation across all states and rating periods. For example, getting a subtotal of premiums policy-wide prior to determine the premium discount percentage.

You can think about this like certain step numbers are synchronization points in a process that is otherwise a parallel calculation by state and rating period. For each combination of state and rating period, each calculation pauses at that step number to perform some calculation across the full policy period. If you are interested in the details, refer to the `WCRatingEngine` method called `rateJurisdictionCosts`.

For worker's compensation rating steps, PolicyCenter sets up a double loop over state and rating period. It checks whether there are any days to rate within the rating period. This is because PolicyCenter may be rating only a premium report for 1 month, so it can skip any rating period that does not overlap with that 1 month.

The `rateJurisdictionCosts` method calls the `processWCRatingSteps` method, which sets up the loop across state and jurisdiction. Within the loop, it calls the `processWCRatingStepsByPeriod` method. That method looks up data from the configuration table and then iterates across each step. It uses a `switch` statement that calls code that knows how to handle each type of step. It defines a number of different types of steps that the engine can use. Many of the calculations that the worker's compensation rating algorithm requires can use a generic handler. For example, calculating a subtotal or applying a modifier. However, some of the most important calculations require specific handlers, such as determining the expense constant or minimum premium.

In workers' compensation, there are standard subtotals that the rating engine uses in later steps. For example, manual premium, modified premium (after applying the experience modification), and so on. The table defines when to calculate each standard subtotal and then which subtotal to use in subsequent calculations.

The most complex code is what happens at synchronization points, particularly for the premium discount. The basic idea is to determine the total standard premium policy wide. Next, use that number to determine the discount percent based on the rules for each state and (rating period start) date independently.

For a premium report, in which you only rate a portion of the period, the rating engine uses the discount percent previously calculated, for example during submission or renewal. The rating engine does not recalculate the discount percent during the premium report because the calculation only makes sense when rating the entire period.

Premium reports for workers' compensation

Remember that the rating process may be for a normal policy transaction, but can also be for a premium report for only part of the period.

A final audit covers the whole period, so from a rating perspective, in most ways it is just like other jobs. There is also a column in the rating steps table called `includeInReport`. If it is `false`, that means to skip this step when doing premium reporting. Use this for steps that a premium report excludes but are part of a final audit.

The worker's compensation implementation for premium reports is in the Gosu enhancement called `WCCoveredEmployeeExt`. For example, the enhancement properties `EffectiveDateForRating` and

`ExpirationDateForRating`. That code determines the overlap between the effective date of the exposure and the audit period.

Inland marine rating

The built-in rating engine for the inland marine line of business is the Gosu class `gw.lob.im.IMRatingEngine`.

The inland marine line of business is unusual because each line consists of multiple coverage parts (`CoveragePart` objects). The industry uses many varied coverage parts, but the supported coverage parts for the default rating engine are listed below.

- Sign part
- Contractor equipment part
- Accounts receivable part

Each of these parts typically rate in very different ways.

The `IMRatingEngine` class overrides the logic of the base class for rating engines and calls out to other classes to do the actual rating for each coverage part. In other words, each coverage part has its own separate class that acts like a rating engine just for that part.

Look in the `IMRatingEngine` class for the private method `rateByPart`. For each slice date, the code first gets the relevant `InlandMarineLine` entity. Next, it iterates across each part and calls a private method called `ratePart` for each `CoveragePart` object. The private method `ratePart` checks the `CoveragePart` subtype and gets the appropriate rating subclass for each type to rate that information. These part-specific rating engine subclasses extend a base class unique for inland marine: `IMAbstractPartRatingEngine`.

Although the name for this class contains the words “rating engine,” `IMAbstractPartRatingEngine` does not extend the rating engine root class `AbstractRatingEngine`. The `IMAbstractPartRatingEngine` is just a special utility class for use only with inland marine rating. This class allows the default rating engine to separate the shared rating behavior from part-specific rating calculations (such as rating the contractor equipment part).

Each of these classes have a `rate` method that does the actual work of rating for inland marine. The `IMRatingEngine` class calls the `rate` method on each part-specific class. The `rate` method returns a list of cost data objects (`List<CostData>`).

General liability rating

The general liability line of business varies from the standard rating algorithm because most general liability costs relate to what are referred to as *exposures*. Like the workers’ compensation line, many general liability classification exposures (also known as *basis*) appear in the user interface in window mode of the full policy term. To properly rate the policy, you may need to specify what portion of the full term exposure applies before and after a rating change. For example if 70% of a store’s sales is in December, you want 70% of the sales exposure to use the rates effective during that month.

Some general liability premiums are determined with rate-scalable costs with effective dates like standard lines of business.

Thus, the rating engine for general liability is a hybrid between the workers’ compensation approach (all costs are unsliced) and standard rating approach (mostly by slice dates).

The structure of general liability data also ensures that the general liability rating engine cannot rate merely from a slice date forward.

The general liability rating engine implements the following basic strategy.

1. The default rating engine determines which exposures are rate scalable. To do this, the default rating engine examines the class code on each exposure. If the class code of the exposure has a `Basis` whose `Auditable` property has the value `false`, then this exposure is rate scalable. Note that the `Auditable` flag is not directly on the class code or the exposure.
2. The default rating engine rates each rate scalable exposure in sliced mode, just like a standard line of business rating engine iterates across slice dates.
3. The default rating engine rates each basis scalable (non-rate-scalable) exposure in window mode, similar to the workers’ compensation rating engine.

4. The default rating engine adds taxes and fee costs in window mode.

Another unusual aspect of general liability rating is that there could be 1, 2, or 4 costs for each coverage, depending on two different factors.

- If the line has its `SplitLimits` property has the value `true`, then the rating engine tracks a bodily injury (PI) limit separate from property damage (PD) limit. Otherwise, the rating engine treats them as a single value called a combined single limit (CSL).
- General liability coverages have cost sublines for both Premises and Products. The general liability rating lookup table for a limit returns two values: one for Premises and one for Products. If it returns a zero for either subline, the default rating engine throws away that cost information. the rating engine may return a non-zero cost for both sublines, or it may return a cost only for one of the two sublines.

Because of a combination of these two factors, each coverage on each exposure can generate 1 cost, 2 costs, or 4 costs.

Be very careful if you modify any code related to general liability or any new similar lines of business. You must ensure that your code tracks the proper number of costs (1, 2 or 4) as described in this topic.

For more details, refer to the implementation class `gw.lob.gl.rating.GLRatingEngine`.

Guidewire Rating Management and PCRatingPlugin

If you use Guidewire Rating Management, you must use a different plugin implementation class than the default rating plugin `SysTableRatingPlugin`. This feature requires that you change the registered plugin implementation for `IRatingPlugin` to the new class `gw.plugin.policyperiod.impl.PCRatingPlugin`.

For the personal auto line of business, this plugin uses the rating engine class `gw.lob.pa.rating.PARatingEngine`. For commercial property, this plugin uses the rating engine class `gw.lob.cp.rating.CPRatingEngine`. For homeowners, this plugin uses the rating engine class `gw.lob.hop.rating.HOPRatingEngine`. For all other lines of business, the plugin calls its superclass `SysTableRatingPlugin` to create the default rating engine instances.

Enable the rating plugin for Guidewire Rating Management

1. In Studio, navigate to **configuration**→**config**→**Plugins**→**registry**, and then open `IRatingPlugin.gwp`. This is the interface definition for the rating plugin.
2. Change the `class` field to `gw.plugin.policyperiod.impl.PCRatingPlugin`.
3. Click **+** to add a `RatingLevel` parameter and set the value to one of the following values.
 - Active
 - Approved
 - Stage
 - Draft
4. Obtain and install a license key for Guidewire Rating Management.

Extending Guidewire Rating Management to other lines of business

To use Rating Management with other lines of business, perform the following operations.

- Write additional rating engine classes using the structure of `PARatingEngine` or `CPRatingEngine` as a guide. You can use personal auto as a guide for personal lines of business and commercial property as a guide for commercial lines of business.
- Make sure that you configure your line of business to instantiate your new rating engine class. In your `PolicyLineMethods` class for your line of business, find the `createRatingEngine` method. This method must return an instance of your new rating engine class.

See also

- “Create a new policy line rating engine” on page 405

Configuring parallel rating

In the base configuration, parallel rating is available in the commercial property line of business. Through configuration, you can extend parallel rating to other lines of business. Parallel rating requires Guidewire Rating Management. Two implementations of parallel rating are provided in the base configuration:

- Parallel rating using entities
- Parallel rating using data transfer objects (DTOs)

See also

- *Application Guide*

Enabling and configuring parallel rating

You can enable parallel rating by setting the `ParallelizedRatingEnabled` configuration parameter in `config.xml`. By default, parallel rating is enabled in the base configuration.

To optimize performance, do benchmark testing to determine the optimal settings for the configuration parameters that control parallel rating. These parameters are:

`ParallelizedRatingEnabled`

Enable or disable parallel rating. If `true`, parallel rating is enabled for each line of business which supports parallel rating. If `false`, parallel rating is disabled.

`MaxRatingThreadPoolSize`

Maximum number of threads spawned during parallel rating. The optimum number of threads depends upon how many tasks the typical policy generates and how long each task takes to complete. The optimum number of threads also depends upon the number of available processors, the typical number of users, and competition for processors from other PolicyCenter features which also implement parallel rating. As a starting point, set the initial value to the number of available processors in your environment. Be aware that other processes or PolicyCenter features which use parallel processing, such as product model availability, may be competing for processors. Perform benchmark testing to better determine the maximum number of threads.

`ParallelRatingTimeoutPerCoverable`

The number of milliseconds per coverable to wait for parallel rating to complete. Because the timeout is per coverable, policies with a large number of coverables will have a longer timeout than smaller policies. This timeout applies to all lines of business that implement parallel rating.

For example, when quoting a policy with 10 coverables and using the default timeout of 20 seconds per coverable, the server times out on rating 200 seconds after it starts. When the timeout is reached, PolicyCenter displays an error to the user and attempts to stop the rating thread.

A value of `-1` implies no timeout and the queue waits until rating completes or the session times out. This is not recommended.

The `ParallelizedRatingEnabled` configuration parameter setting enables parallel rating for the whole PolicyCenter implementation. In addition, each line can implement and enable parallel rating.

`PolicyLine#shouldParallelizeRating` is set globally `false` in `AbstractPolicyLineMethodsImpl.gs`. To enable parallel rating in a policy line, override the global setting by setting `shouldParallelizeRating` to `true` in the `XXPolicyLineMethods` class. You can also configure criteria for when to use parallel rating in the `shouldParallelizeRating` method.

In the base configuration, parallel rating is enabled for commercial property. The `CPPolicyLineMethods` class, `shouldParallelizeRating` returns `true`.

Additional parameter for parallel rating using DTOs

To enable parallel rating using DTOs for commercial property, you must enable the following parameter:

- `EnableCPDTONParallelRating` – Enable parallel rating using data transfer objects (DTOs) for commercial property. This parameter also enables access to Rating Management components in the sample data, such as rate table and rate routines, that use DTOs. These components are included in the small sample data set and are provided for demonstration purposes.

Implementing parallel rating

Through configuration, you can implement parallel rating in a line of business. You can use commercial property, which implements parallel rating, as a model.

Can this policy line be rated in parallel?

Consider parallel rating for lines of business with policies that typically include a large number of coverables, and where generating quotes takes a noticeable amount of time. Parallel rating iterates over coverables in parallel using multiple threads to generate costs for each of the coverables.

The coverable object that will be rated in parallel must meet the following requirements:

- The coverable must be `EffDated`
- The coverable must not share data with other coverables that will be rated in parallel, or
- Data shared among the coverables must be immutable or accessed in a thread-safe manner

Consider the size of the coverable when deciding which coverable to rate in parallel. If the coverable is large, then parallel rating requires fewer threads but may result in threads waiting idly for threads to finish rating other large coverables.

Ensuring thread safety

Because parallel rating makes use of multiple threads, the code must be thread-safe.

The implementation of parallel rating uses the following general coding guidelines for thread safety:

- The `PolicyLine` variable is `ThreadLocal` so that each thread has its own instance of the `PolicyLine`.
- Because bundles are not thread-safe, each thread works on its own separate bundle, and this bundle does not write to the database.
 - For parallel rating using entities, access to in-memory objects related to rating, including `CostDataMap` and `WorksheetContainer`, must be thread-safe.
 - For parallel rating using DTOs, the rating threads do not have access to a bundle. Consequentially, access to `CostDataMap`, `WorksheetContainer`, and other entities is deferred until rating is complete.

The `AbstractParallelRatingEngineBase` class implements thread safety, and the `CPRatingEngine` class calls the methods in this class.

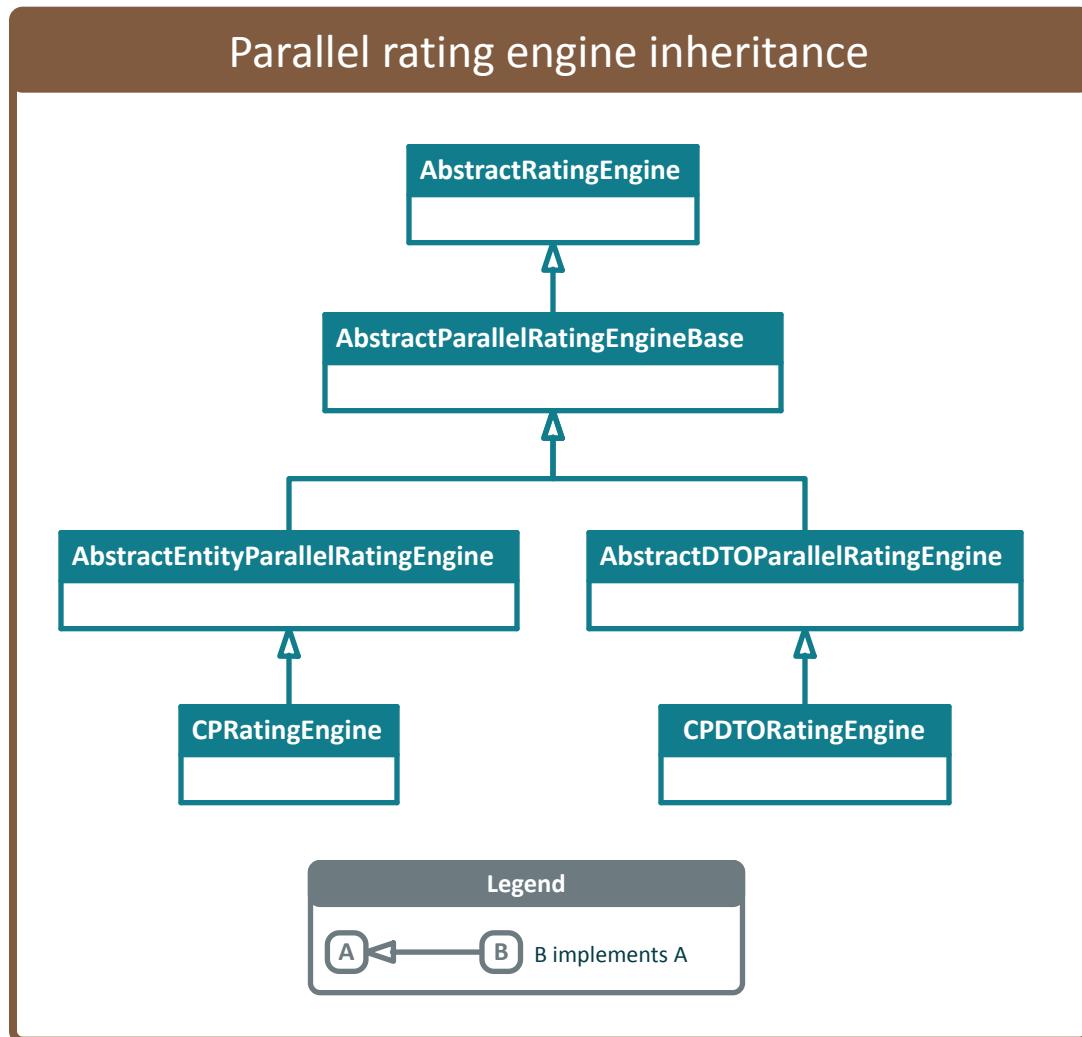
- The rating engine code has been reordered for thread-safety. Some tasks have been moved to the `preRateStep` method which is called before the code which runs in parallel. Other tasks have been moved to `postRateStep` method which is called after the code which runs in parallel.
- The rating engine for the line of business calls the `rateInParallel` method, passing in a list of coverables along with the block of code that rates each coverable.
- The `rateInParallel` method, spawn threads. Each thread rates a coverable by executing the block of code on the coverable. These threads are managed by a thread pool.
- For parallel rating using entities, if the server is in development mode, the `checkOriginalBundleState` method verifies that the original coverables have not been modified.

Logging

Parallel rating logs general messages at `INFO` level and more detailed messages at the `DEBUG` level. Logging at the `INFO` level includes general information about actions taken by the rating engine. Logging at the `DEBUG` level includes timing for the overall rate as well as details about how long individual threads took to rate coverables.

Overview of classes that implement parallel rating

The rating engine and related classes contain the code that implements parallel rating using entities and parallel rating using DTOs. The following illustration shows object inheritance for both implementations.



Rating engine using entities

For parallel rating using entities, the policy line implementation of the rating engine, `XXRatingEngine` extends `AbstractEntityParallelRatingEngine`.

For commercial property, the rating engine is `CPRatingEngine`. You can use this as a reference implementation for adding parallel rating to a line of business.

Rating engine using DTOs

For parallel rating using DTOs, the policy line implementation of the rating engine, `XXDTORatingEngine` extends `AbstractDTOParallelRatingEngine`.

For commercial property, the rating engine is `CPDTRatingEngine`. You can use this as a reference implementation for adding parallel rating using DTOs to a line of business.

Policy line methods

For both types of parallel rating, `AbstractPolicyLineMethodsImpl` implements policy line methods and contains the Boolean `shouldParallelizeRating` method. Each line of business can override this value in `XXPolicyLineMethods` class.

For commercial property, `CPPolicyLineMethods` overrides this method to always returns `true`. Modify this method to specify criteria for parallel rating of commercial property policies.

Abstract parallel rating engine

For both types of parallel rating, the `AbstractParallelRatingEngineBase` class encapsulates the code that implements parallel rating, and contains the `rateInParallel` method.

`AbstractParallelRatingEngineBase` rates serially instead of in parallel when either of the following is true:

- The `ParallelizedRatingEnabled` configuration parameter is `false`
- `XXPolicyLineMethods#shouldParallelizeRating` returns `false`

Extend parallel rating using entities to another line of business

These instructions provide general guidance.

About this task

The rating engine that you modify in the following steps is `XXRatingEngine.gs`, where `XX` is the code for the line of business. For example, `CPRatingEngine.gs` is the rating engine for commercial property.

Procedure

1. In your implementation of `XXPolicyLineMethods.gs` for the line of business, override the `PolicyLineMethods#shouldParallelizeRating` method to return `true`, or return `true` when other conditions are met. For example, if you wish to limit parallel rating to policies with a large number of coverables, add that logic to this method.
2. Modify the rating engine for the line of business to extend `AbstractEntityParallelRatingEngine` instead of `AbstractRatingEngine`.

For example, `CPRatingEngine.gs` class is declared as:

```
class CPRatingEngine extends AbstractEntityParallelRatingEngine<CPLine, EffDated>
```

3. Modify the `rateSlice` method in the rating engine for parallel rating. In `rateSlice`:
 - a. Call the `preRateStep` method. In the base configuration, this method is defined in `AbstractParallelRatingEngineBase`. If you need to customize this method for this line of business, add a `preRateStep` method to `XXRatingEngine`.
 - b. Call the `rateInParallel` method and pass in the list of coverables to rate in parallel and the code block that rates each coverable.
 - c. Call the `handleFutures` method on the list of `Futures` returned from `rateInParallel`.
 - d. Call the `postRateStep` method. In the base configuration, this method is defined in `AbstractParallelRatingEngine`. If necessary, customize as for `preRateStep`.

Extending parallel rating using DTOs to other lines of business

You can extend parallel rating using DTOs to lines of business that are currently using entities. You must first modify the line for parallel rating.

You can create new lines of business that have parallel rating using DTOs. You must implement the new line with both parallel rating and DTOs.

See also

- *Application Guide*

Considerations

When defining the DTOs, keep in mind the following:

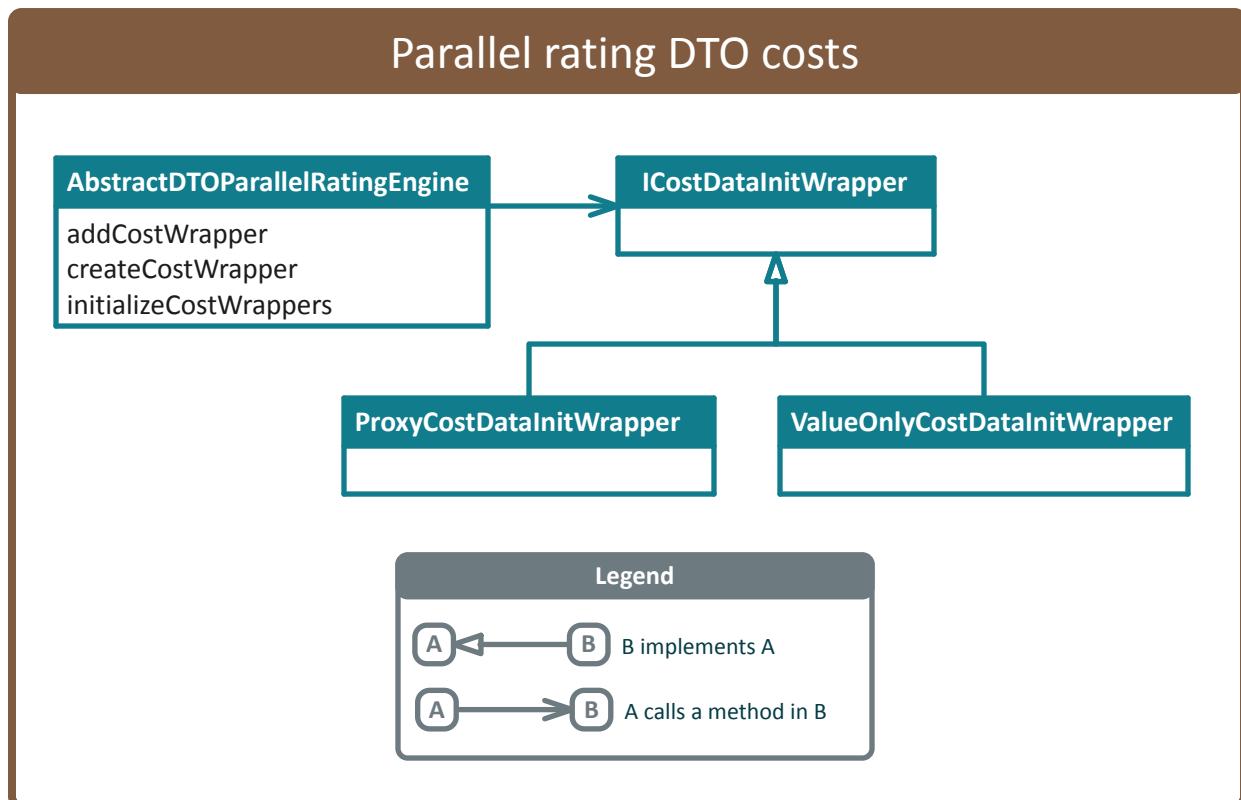
- Some properties, easily accessed through parameters in Rating Management are not as easily discovered in the data model. In commercial property for example, the `PolicyLine` parameter has a `CPScheduleCredits` field. In the data model however, `CPScheduleCredits` is not property of `PolicyLine`. It is the line's `RateModifier` which matches the `CPScheduleCredits` pattern code. This kind of nuance is hidden from the user in the rate routine and requires investigation to figure out what data actually needs to be stored in the DTO.
- Some Rating Management parameters have a value provider, so your DTO must exactly match the data type of the value provider. In commercial property for example, the `BaseRate` rate table's `COV_CODE` parameter uses the Coverage Value Provider and only accepts data from a coverage. One workaround is to make a new rate table definition without value providers so the `COV_CODE` parameter accepts a string from the DTO.
- The parameter set must include the `PolicyLine` entity because Rating Management code accesses it.
- If you use rate routines that use entities, this rating must be done outside the `rateInParallel` method. In commercial property for example, rating a policy uses the Generic State Tax Calculation rate routine, which is generic to all lines of business and uses entities.

Rating engine for parallel rating using DTOs

The `AbstractDTOParallelRatingEngine` is the base class for DTO rating engines. This class creates the parallel rater and provides methods that create and initialize access to the information in `Cost` and `CostData` objects.

Cost wrappers

Parallel DTO rating uses various classes for accessing `Cost` and `CostData` objects.



The `ICostDataInitWrapper` interface replaces many of the `Cost` references in the `CostData` and `RatingEngine` classes. Implementations of this interface perform the same functionality as the `init` methods in the `CostData` classes. There are two implementations of `ICostDataInitWrapper`:

ProxyCostDataInitWrapper

Acts as a proxy receiving a reference to a `Cost` entity. Create methods that retrieve property values from the entity. Use this proxy to maintain the logic in an entity-based parallel rating implementation. In the base configuration, parallel rating using entities implements this wrapper. This cannot be used by parallel rating using DTOs.

ValueOnlyCostDataInitWrapper

Copies all property values from a `Cost` object so that parallel rating can access these values without requiring access to the entity. In the base configuration, parallel rating using DTOs implements this wrapper.

Initializing cost wrappers

Any rating engine extending `AbstractDTOParallelRatingEngine` must implement the `initializeCostWrappers` method. This method populates a map of `ICostInitDataWrapper` objects containing all `CostData` information. The `preRateStep` method in the parent threads accesses `CostData` indirectly through these wrappers.

Handling CostData created by the rating engine

During parallel rating, `CostData` objects generated by child threads are added to a temporary map, called the staging map. In `AbstractDTOParallelRatingEngine`, see the `addStagedCost` and `addStagedCosts` methods. After rating, the `postRateStep` method applies the staged `CostData` objects to the `CostDataMap` for the policy line.

Extend parallel rating using DTOs to a line of business

These instructions provide general guidance.

Before you begin

Convert the line of business to parallel rating. See *Configuration Guide*.

About this task

You can view Gosu classes that implement DTOs for commercial property in Studio in the `gw.api.rating.dtobased.data.cp` package. This code makes use of read-only DTO classes that provide basic DTO implementations for entities commonly used in policies. These classes include `CoverageDTO` and `EffDatedDTO` among others. To access these classes, open a commercial property example, such as `CPBuildingCovDTO`. This class extends `CoverageDTO`. Press `Ctrl` and click `CoverageDTO` to open that class.

Procedure

1. For each rate routine note which of its parameters are entities and which entity fields from each of those entities are needed for the calculation. Each DTO needs to contain the fields of its corresponding entity that are used as operands in the rate routine and as arguments for table lookups.

Define DTO classes

2. In `gw.api.rating.dtobased.data.Lob`, where `Lob` is the abbreviation for the line of business, make a DTO Gosu class for each entity that is being replaced.
3. In the Gosu class, add the necessary variables to store the entity data. Numbers that are used for calculation can be represented as `BigDecimal` and fields that are used for table lookups can be represented as `String`.
4. Add a typecode for each DTO in `CalcRoutineParamName.ttx`.

Rating engine

5. Using `CPDTRatingEngine` as a guide, make changes to the parallel rating engine to initialize the DTOs and load their data from the entities.
6. Modify each rate routine call replacing the parameter map with a map that includes the DTOs, referenced by their typecodes, and the `PolicyLine` entity.

The rating engine calls the rate routine using the `RateBook.executeCalcRoutine` method.

The `PolicyLine` entity must be in the parameter set and is required by internal rating code.

In PolicyCenter

7. In PolicyCenter, create a new parameter set that includes the DTOs, the `PolicyLine` entity, and any other parameters from the original parameter set that were not replaced by DTOs.
8. If necessary, create new rate table definitions to allow data from the DTO as an argument for rate table lookup.
In commercial property for example, the `BaseRate` rate table definition (`cp_coverage_base_rate_dto`) was reimplemented because it specified value providers for `COV_CODE` and `CAUSE_OF_LOSS`. The new rate table definition is identical but does not specify any value providers.
9. Create new versions of rate routines that use the new parameter set and new rate table definitions. Replace entity references with DTOs.
10. Create a new version of the rate book that includes the new rate routines and rate table definitions.
11. For each new rate table definition, you must newly import the rate table data.
 - a. Export the new DTO rate table to spreadsheet format to get the formatting for the data.
You can now view and edit the content in Microsoft Excel.
 - b. Export the old rate table to spreadsheet format.
See *Application Guide*.
 - c. Copy the relevant columns to the spreadsheet for the new rate table.
 - d. Import this spreadsheet into the new rate table.
 - e. Now the rate book can be approved and promoted.

Reinsurance integration

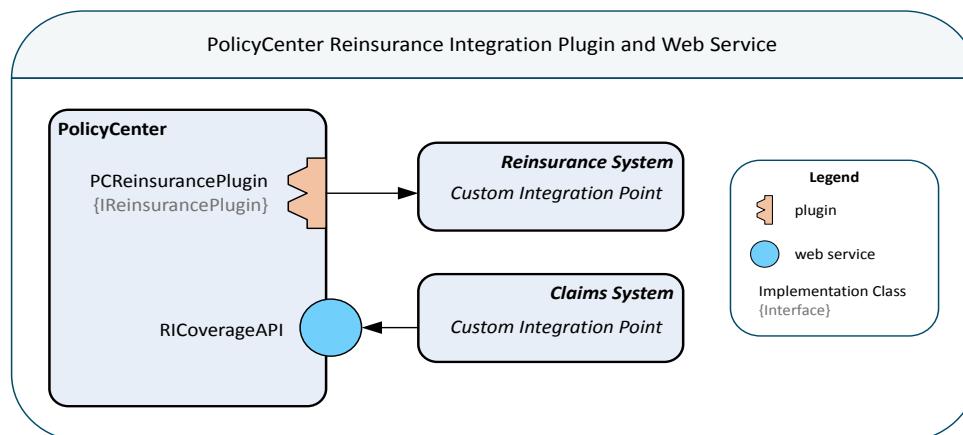
PolicyCenter supports reinsurance handling through integration of Guidewire Reinsurance Management or your own reinsurance management system.

Note: Guidewire Reinsurance Management is available within Guidewire PolicyCenter. To determine whether your Guidewire PolicyCenter license agreement includes Reinsurance Management, contact your Guidewire sales representative. Reinsurance Management requires an additional license key. For instructions on obtaining and installing this key, contact your Guidewire support representative.

Reinsurance is insurance risk transferred to another insurance company for all or part of an assumed liability. In other words, reinsurance is insurance for insurance companies. When a company reinsures its liability with another company, it cedes business to that company. The amount an insurer keeps for its own account is its retention. When an insurance company or a reinsurance company accepts part of another company's business, it assumes risk. It thus becomes a reinsurer.

The insurance company directly selling the policy is also known in the industry as the *insurer*, the *reinsured*, or the *ceding company*. The Guidewire term for this company that directly sells the policy is *insurer*. An insurance company accepting ceded risks is known as the *reinsurer*.

The following diagram illustrates the reinsurance plugin and web service that PolicyCenter provides for integration with external systems.



The plugin lets you integrate a reinsurance system with PolicyCenter. The web service lets you integrate PolicyCenter with a claims system for the purposes of reinsurance coverage.

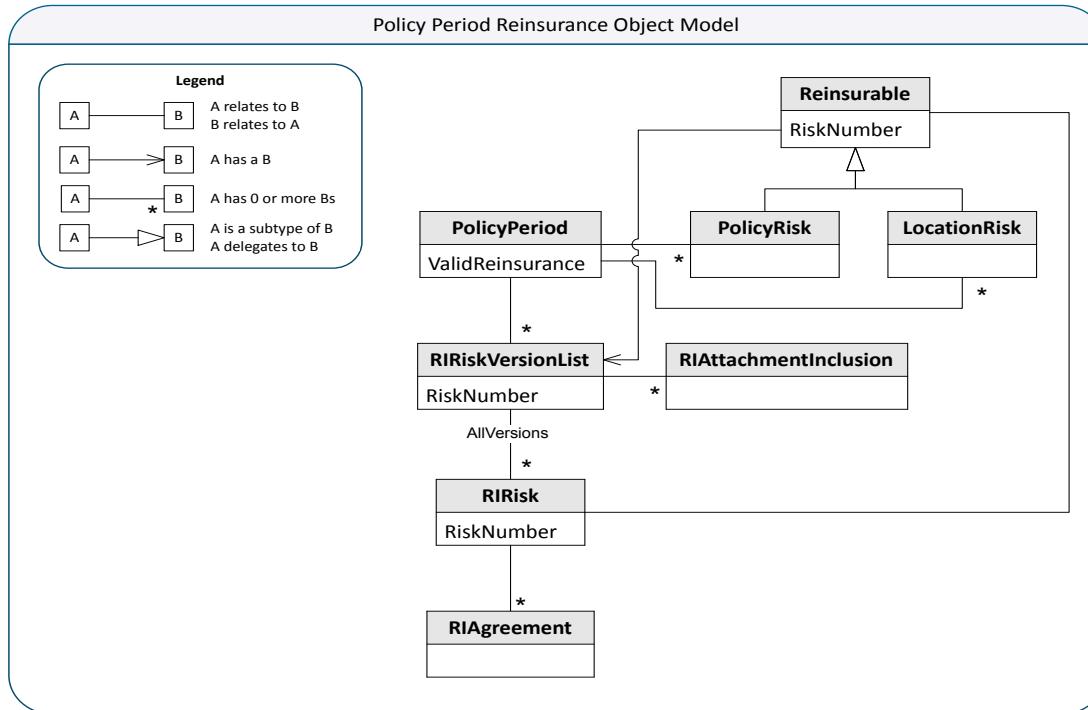
- **IReinsurancePlugin** – Plugin that lets you integrate a reinsurance system with PolicyCenter. The reinsurance system can be Guidewire Reinsurance Management, or it can be your own reinsurance system.
- **RICoverageAPI** – Web service that lets claims systems, such as Guidewire ClaimCenter, retrieve reinsurance coverage information from policies in PolicyCenter.

See also

- *Application Guide*
- *Configuration Guide*

Reinsurance data model

The PolicyCenter reinsurance plugin and web service use several entities in the PolicyCenter data model. The following diagram illustrates how reinsurance entities relate to policy periods.



A **PolicyPeriod** has a **ValidReinsurance** property that flags policies with reinsurable risks (**Reinsurable** and descendants). Rules or people authorized as reinsurance managers determine which policy periods are valid for reinsurance and what specific reinsurable risks to attach.

Each reinsurable risk has a unique **RiskNumber** and a list of **RIRisk** instances (**RIRiskVersionList**). Each **RIRisk** instance in a **RIRiskVersionList** represents the details of a reinsurable risk for one interval of effective time on a policy period.

Risk entity

The **RIRisk** entity represents a reinsurable risk on policy period. If a policy period qualifies for reinsurance coverage, each reinsurable risk has a list of **RIRisk** instances.

Effective dates on risk instances

The **RIRisk** entity has effective dates. All **RIRisk** instances have **EffectiveDate** and **ExpirationDate** properties, because the **RIRisk** entity is an implementation of a **SimpleEffDated** entity. An **RIRisk** instance represents the details of one reinsurable risk for one interval of effective time on a policy period. Multiple **RIRisk** instances can represent the same reinsurable risk on a policy period for different, non-overlapping intervals.

RIRisk instances for the same reinsurable risk have the same value for their **RiskNumber** properties. Each instance one represents the details of the reinsurable risk during different effective date ranges. Effective date ranges must abut and not overlap. Together, the instances represent a timeline of changes to the reinsurable risk, during the policy period. When the details of a reinsurable risk change, PolicyCenter splits the active **RIRisk** in two. The original and new **RIRisk** instances have different effective date ranges that abut and do not overlap.

Reasons other than changes to a reinsurable risk cause PolicyCenter to split **RIRisk** entities in two. For example, an instance of reinsurance entity that links to an instance of an **RIRisk** changes. As a result, PolicyCenter splits all related reinsurance instances in two that are in effect on the effective date of the change, including the **RIRisk**.

Multiple **RIRisk** instances with the same **RiskNumber** represent one reinsurable risk on a policy period. **RIRisk** instances with the same **RiskNumber** must have effective date intervals that abut and do not overlap.

Properties of risk instances

The **RIRisk** entity has a number of properties. The important properties for the reinsurance plugins and web service are listed below.

Method	Description
Agreements	Array of ReinsuranceAgreement instances associated with this risk.
Attachments	Array of ReinsuranceAttachment instances associated with this risk.
Reinsurable	Foreign key to the reinsurable risk (a subtype of Reinsurable) that this RIRisk describes.

Methods on risk instances

The **RIRisk** entity has a number of methods. The methods for the reinsurance plugins and web service are described below.

Method	Description
attach	Attaches this RIRisk to a reinsurance agreement. The attach method takes a reinsurance agreement (RIAgreement) and a reinsurance program (RIProgram). A reinsurance agreement can be belong to several reinsurance programs.
canAttach	Determines whether you can attach a given ReinsuranceAgreement to this RIRisk . If errors that prevent attachment are not found, then the canAttach method returns an empty list, not null. If errors that prevent attachment are found, then the canAttach method returns a list of the errors as human-readable String values.
detach	Detaches a risk or policy attachment from an agreement. The detach method has two signatures. One takes an RIRisk , and the other takes an RIPolicyAttachment .
makeActive	Makes this RIRisk of the reinsurable risk the active version. The active version is the RIRisk is the one that is in effect today.

Risk version list entity

The **RIRiskVersionList** entity helps manage multiple versions of a single reinsurable risk on a policy period. An **RIRiskVersionList** holds all the **RIRisk** instances for a single reinsurable risk, in an array called **AllVersions**. **RIRisk** instances have a foreign key called **VersionList** that links them back to the **RIRiskVersionList** instance that holds them. An **RIRiskVersionList** has the same value for its **RiskNumber** property as the **RIRisk** instances that it manages.

An `RIRisk` instance relates to its policy period through the `RIRiskVersionList` instance that manages it. An `RIRiskVersionList` has a foreign key to its policy period, called `PolicyPeriod`. A `PolicyPeriod` instance has a derived array of its reinsurable risks, called `AllRIRisks`. If a `PolicyPeriod` has any related `RIRiskVersionList` instances in its `AllRIRisks` array, its `ValidReinsurance` property is `true`.

Properties of risk version lists

The `RIRiskVersionList` entity has a number of properties. The properties related to the reinsurance plugins and web service are described in the following table.

Property	Description
<code>RiskNumber</code>	Unique identifier of a reinsurable risk.
<code>AllVersions</code>	Array of <code>RIRisk</code> instances that represent different versions of the same reinsurable risk.
<code>AttachmentInclusions</code>	<code>PolicyCenter</code> creates an <code>RIAttachmentInclusion</code> if the inclusion status of the attachment differs from the default, <code>Included</code> . Therefore, only attachments that are excluded or have special acceptance have an attachment inclusion row. When you exclude an attachment, <code>PolicyCenter</code> creates an <code>RIAttachmentInclusion</code> with the status set to <code>Excluded</code> .
<code>PolicyPeriod</code>	Policy period to which all versions of a reinsurable risk provide reinsurance coverage. Each version covers a different effective date range that does not overlap with the others. Only one version is active at a time, though sometimes none are active.

Methods on risk version lists

The `RIRiskVersionList` entity has a number of methods. The methods related to the reinsurance plugins and web service are described in the following table.

Method	Description
<code>addToAttachmentInclusions</code>	Adds an <code>RIRisk</code> to the array of <code>RIAttachmentInclusion</code> instances on this <code>RIRiskVersionList</code> . The array name is <code>AttachmentInclusions</code> . An <code>RIAttachmentInclusion</code> relates an <code>RIRisk</code> on a policy period to a reinsurance agreement.
<code>addVersion</code>	Adds an <code>RIRisk</code> to the array of <code>RIRisks</code> on this <code>RIRiskVersionList</code> . The array name is <code>AllVersions</code> .
<code>endDate</code>	Splits the active <code>RIRisk</code> by doing all of the following operations. <ul style="list-style-type: none"> • Sets <code>ExpirationDate</code> to today on the current <code>RIRisk</code>. • Creates a new <code>RIRisk</code> instance with the same <code>RiskNumber</code> as the original. • Sets the <code>EffectiveDate</code> on the new instance to today. • Replaces all remaining fields except <code>ExpirationDate</code> from the original. • Adds the new instance to the same <code>RIRiskVersionList</code> as the original.
<code>getRIRisk</code>	Gets the <code>RIRisk</code> that is active today.
<code>getVersionAsOfDate</code>	Gets the <code>RIRisk</code> that is active for a specified date.
<code>removeFromAttachmentInclusions</code>	Removes an <code>RIRisk</code> from the array of <code>RIAttachmentInclusion</code> instances on this <code>RIRiskVersionList</code> . The array name is <code>AttachmentInclusions</code> . An <code>RIAttachmentInclusion</code> relates an <code>RIRisk</code> on a policy period to a reinsurance agreement.

Reinsurance plugin

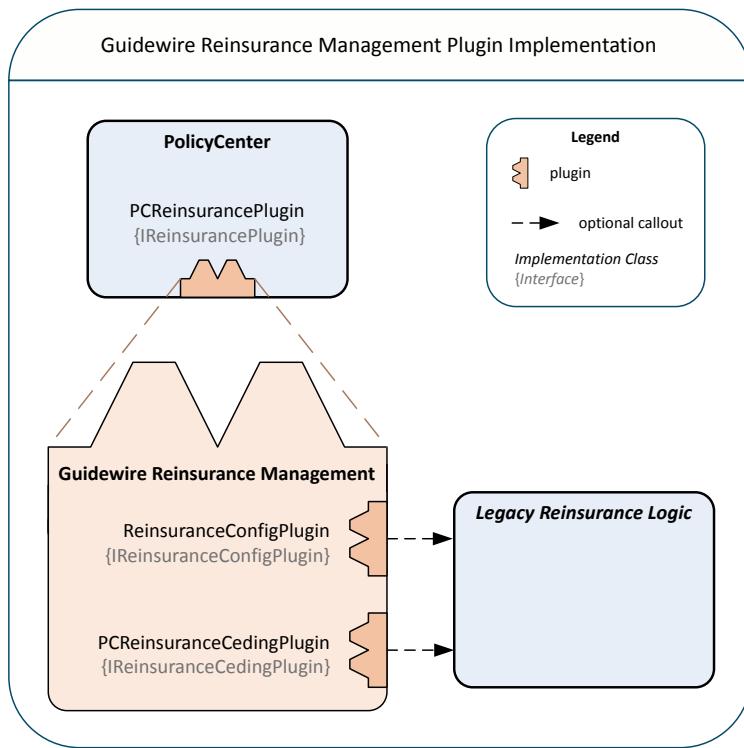
PolicyCenter provides the `IReinsurancePlugin` plugin interface to integrate with Guidewire Reinsurance Management or let you integrate your own reinsurance management system with PolicyCenter. You can find the

Plugins registry for the `IReinsurancePlugin` in PolicyCenter Studio by navigating in the **Project** window to **configuration**→**config**→**Plugins**→**registry**. The source code is in the `IReinsurancePlugin.gwp` file.

In the base configuration of PolicyCenter, the Gosu class `PCReinsurancePlugin` is a implementation of the `IReinsurancePlugin` plugin interface. The `PCReinsurancePlugin` enables the default behaviors of Guidewire Reinsurance Management within PolicyCenter.

Architecture of the Reinsurance Management plugin

Guidewire Reinsurance Management screens in the PolicyCenter user interface call the registered plugin in response to actions by users authorized as reinsurance managers. In the base configuration of PolicyCenter, the Plugins registry includes the `PCReinsurancePlugin` Gosu class as an implementation of the `IReinsurancePlugin` plugin interface. As shown in the following illustration, this plugin integrates Guidewire Reinsurance Management with PolicyCenter.



You might want to configure Guidewire Reinsurance Management with your own logic for assembling treaties into programs, assigning treaties to policies, and calculating ceded premiums. For example, you extend the PolicyCenter data model for reinsurance with new entities or properties. So, you must configure the `PCReinsurancePlugin` plugin implementation with your own logic to handle your data model extensions.

To integrate your own reinsurance system with PolicyCenter, you must write your own implementation of the `IReinsurancePlugin` plugin interface. The default Gosu implementation `PCReinsurancePlugin` demonstrates internal details of the methods that you must provide in your own implementation of `IReinsurancePlugin`.

Configuration plugins for Guidewire Reinsurance Management

The `PCReinsurancePlugin` implementation of the reinsurance plugin uses the following configuration plugins to let you enhance Guidewire Reinsurance Management with your own reinsurance logic.

- `IReinsuranceConfigPlugin` – This plugin interface lets you integrate your own reinsurance logic to control how PolicyCenter configures reinsurable risks on specific policies. The Gosu class `gw.plugin.reinsurance.ReinsuranceConfigPlugin` implements this interface.
- `IReinsuranceCedingPlugin` – This plugin interface lets you integrate your own reinsurance logic to perform ceding amount calculations. The Gosu class `gw.plugin.reinsurance.PCReinsuranceCedingPlugin` implements this interface.

In Studio, you can modify the code of these plugins directly, or you can define subclasses of the plugin implementations to override or extend specific methods. You can implement your own logic entirely in Gosu, or you can implement part of your logic in Gosu and make calls out to reinsurance logic embedded in legacy systems. For example, you might enhance Guidewire Reinsurance Management to interact with legacy systems that generate unique risk IDs, store reinsurance data, or send ceded premiums to accounts payable.

To enhance Guidewire Reinsurance Management with your own reinsurance logic, do not modify `PCReinsurancePlugin.gs` directly. Instead, modify the additional plugins mentioned above. In Studio, you can modify the code directly, or you can define subclasses of the additional plugin implementations to override or extend specific methods.

Creating a plugin implementation for your own reinsurance system

If you want to integrate your own reinsurance system with PolicyCenter in place of Guidewire Reinsurance Management, you must develop your own implementation of the `IReinsurancePlugin` plugin. The default implementation `PCReinsurancePlugin.gs` demonstrates internal details of the methods that you must provide in your own implementation of `IReinsurancePlugin`. In addition, you may need to develop or modify rules and user interface components to let users authorized as reinsurance managers handle reinsurance in PolicyCenter.

If you develop your own implementation of `IReinsurancePlugin`, do not develop your own implementations of the plugins `IReinsuranceConfigPlugin` and `IReinsuranceCedingPlugin`. These two plugins are part of Guidewire Reinsurance Management.

Reinsurance plugin implementation

PolicyCenter provides `PCReinsurancePlugin.gs` as an implementation of the `IReinsurancePlugin` plugin interface. This plugin integrates Guidewire Reinsurance Management with PolicyCenter.

In the `gw.job.QuoteProcess.QuoteProcess` class, the `handleValidQuote` method calls the reinsurance plugin. In the base configuration, this plugin is `PCReinsurancePlugin` in the `gw.plugin.reinsurance` package.

Selecting a reinsurance program with the program finder interface

The `PCReinsurancePlugin.gs` plugin calls methods on an implementation of the `RIProgramFinder` interface, `RIProgramFinderImpl`, to select a reinsurance program for the current policy period. For example, the following code gets the `finder` object, which is an instance of `RIProgramFinderImpl`.

```
var finder = PCDependencies.getRIProgramFinder()
```

Programs for succeeding years may either be in draft status or not yet entered into PolicyCenter. The Java class `RIProgramFinderImpl` selects the program for a risk in the following order.

1. Active program for the date range and coverage group.

In the base configuration, the reinsurance plugin returns an error message if it finds more than one matching program.

However, it is relatively common for an insurer to have programs based on geography, such as by country or state, or for risks in cities or other high concentration areas. You can add extra selection logic to the plugin so that it chooses the one correct program.

2. Draft program for the date range and coverage group.

In the base configuration, the reinsurance plugin can return one match.

3. Prior year active program for the coverage group.

The reinsurance plugin finds the most recent program that applies to this risk. Because the plugin is trying to find a match in a date range, there could be several programs that apply. The plugin selects the most recent program.

4. The reinsurance plugin does not find a match.

PolicyCenter writes an error to the log file, but does not block progress on the policy.

If PolicyCenter selects a draft or prior year program, the agreements from that program are marked as **Projected** in the policy.

You can specify other criteria for selecting the program by modifying the plugin. For example, you can modify the plugin to select a program based on the risk location.

When the policy period is quoted, the reinsurance plugin creates or updates the version lists associated with each reinsurable risk in that policy period.

Reinsurance plugin interface methods

The main purpose of the `IReinsurancePlugin` interface is to create and manage `RIRisk` instances associated with reinsurable risks (`Reinsurable` and descendants). PolicyCenter uses the plugin methods as integration points with a reinsurance system, such as Reinsurance Management. If you are integrating your own reinsurance system, you must create your own implementation of the `IReinsurancePlugin`.

Attaching reinsurable risks to policy periods

PolicyCenter calls the `attachRisk` method on `IReinsurancePlugin` to attach the reinsurable risk to the applicable reinsurance agreements. The method splits the risk at program boundaries, if necessary.

```
override function attachRisk(reinsurable : Reinsurable)
```

The `attachRisk` method takes a `Reinsurable` instance.

In the base implementation, the method either finds an `RIRiskVersionList` or creates a new one by copying an older version. The method finds or creates an `RIRisk` as of that date.

Next, the method computes the set of attachments for the `RIRisk`. This method finds applicable reinsurance programs that are effective on a specific date for given coverage group. The default implementation of the reinsurance plugin calls the `finder.findApplicablePrograms` method. This method returns an array of `RIProgram` entities. If you need to modify how PolicyCenter finds applicable programs, you can override this method.

In PolicyCenter, you can see the agreements attached to the policy on the **Reinsurance** screen. If there are multiple programs that apply during the policy period, the **View As Of** drop-down list has multiple date ranges.

See also

- *Application Guide*

Reattaching risks

PolicyCenter calls the `reattachRisk` method on `IReinsurancePlugin` to recompute program-related information associated with the given `RIRisk`. This normally occurs in situations where the reinsurance programs changed. The `reattachRisk` method reattaches a risk to its attachments.

```
override function reattachRisk(reinsurable : Reinsurable)
```

The `reattachRisk` method takes one argument, which is the risk that needs attachments refreshed. It computes the attachments and returns no values.

Removing RIRisk instances associated with reinsurable risks

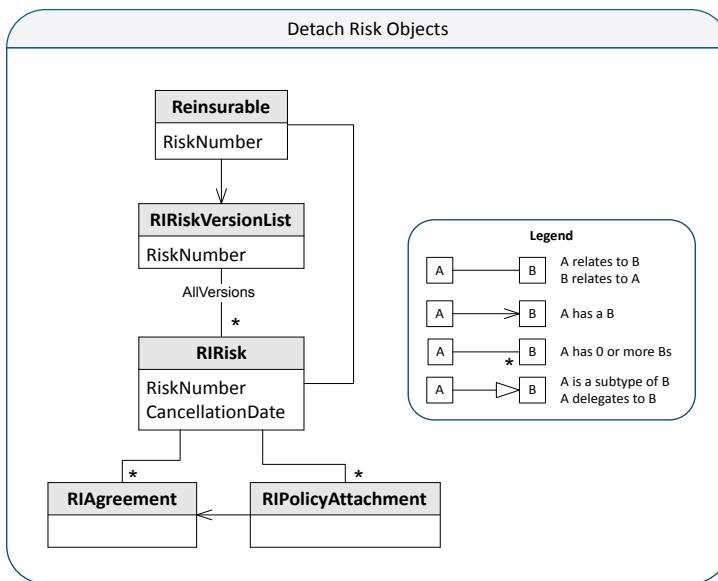
PolicyCenter calls the `detachRisk` method on `IReinsurancePlugin` to remove from the version list a risk that expires after the specified edit effective date of the policy period.

```
override function detachRisk(reinsurable : Reinsurable, branch : PolicyPeriod)
```

In the default implementation for a given `Reinsurable`, this method performs the following operations to each `RIRisk` in the `RIRiskVersionList`.

- Finds the one `RIRisk` that is effective at the edit effective date of the policy period.
- Changes the `ExpirationDate` of that `RIRisk` to the edit effective date of the policy period.
- Removes any `RIRisk` after that date.

In effective time, this means that the `RIRisk` expires on the given date. There is no future version of that risk.



Validating risks

PolicyCenter calls the `validateRisk` method on `IReinsurancePlugin` to validate all risks attached to a reinsurable at the specified validation level.

```
override function validateRisk(reinsurable : Reinsurable, level : ValidationLevel) : EntityValidation[]
```

See also

- *Configuration Guide*

Cleaning up after removing reinsurables in a renewal job

PolicyCenter calls the `postApplyChangesFromBranch` method on `IReinsurancePlugin` to clean up after removing reinsurables from the branch in a renewal job. In a renewal job, PolicyCenter removes the `Reinsurable` entity instances, but the cloned branch may contain implementation data related to the removed `Reinsurable` entity instances.

```
override function postApplyChangesFromBranch(policyPeriod : PolicyPeriod)
```

See also

- “Policy period plugin” on page 168

Binding draft reinsurance version lists associated with a branch

When a policy period is bound, PolicyCenter calls the `bindBranch` method on `IReinsurancePlugin` to bind its associated `RIRiskVersionLists`, one for each reinsurable risk.

```
override function bindBranch(branch : PolicyPeriod)
```

The method must bind all of the draft version list instances associated with a branch. PolicyCenter usually calls this method while binding a policy period.

Withdrawing reinsurance version lists associated with a branch

PolicyCenter calls the `withdrawBranch` method on `IReinsurancePlugin` to withdraw all of the `RIVersionList` instances associated with a branch. PolicyCenter usually calls this method while withdrawing a policy period.

```
override function withdrawBranch(branch : PolicyPeriod)
```

Whether contacts can be deleted

PolicyCenter calls the `isContactDeletable` method on `IReinsurancePlugin` to check if a contact is unused by reinsurance.

```
override function isContactDeletable(contact : Contact) : boolean
```

This method takes a contact as a parameter. The method returns `true` if the contact can be deleted. Otherwise, the method returns `false`.

Getting the location risk group

PolicyCenter calls the `getLocationRiskGroup` method on `IReinsurancePlugin` to return the location risk group for a given reinsurable risk.

```
override function getLocationRiskGroup(risk : Reinsurable) : String
```

The `risk` parameter is the Reinsurable risk from which to get the location risk group. The method returns the location risk group as a `String`.

Setting the location risk group

PolicyCenter calls the `setLocationRiskGroup` method on `IReinsurancePlugin` to set the risk group on the given reinsurable risk.

```
override function setLocationRiskGroup(risk : Reinsurable, locationRiskGroup : String)
```

The `risk` parameter is the Reinsurable risk on which to set the location risk group. The method sets the location risk group on the `RIRisk` associated with the Reinsurable.

Getting risks in a location risk group

PolicyCenter calls the `getRisksInALocationRiskGroup` method on `IReinsurancePlugin` to get the risks in a location risk group at a given date.

```
override function getRisksInALocationRiskGroup(locationRiskGroup : String, date : Date) : List<String>
```

The `locationRiskGroup` is the location risk group from which to get the risks. The `date` is the date on which the risks must be in effect. This method returns all the risks in the group as a list of `riskNumbers`.

Reinsurance configuration plugin

PolicyCenter provides the `IReinsuranceConfigPlugin` to help you configure Reinsurance Management with your own logic to configure policies with reinsurable risks. You can find the registry for the `IReinsuranceConfigPlugin`

in Guidewire Studio by navigating in the **Project** window to **configuration**→**config**→**Plugins**→**registry**. The source code is in the `IReinsuranceConfigPlugin.gwp` file.

Reinsurance configuration plugin implementations

PolicyCenter provides `ReinsuranceConfigPlugin.gs` as an implementation of `IReinsuranceConfigPlugin`. To configure Reinsurance Management with your own logic to configure policies for reinsurance, modify `ReinsuranceConfigPlugin.gs`. In Studio, you can modify the code directly, or you can define a subclass of `ReinsuranceConfigPlugin` to override or extend specific methods.

Reinsurance configuration plugin methods and properties

The main purpose of `IReinsuranceConfigPlugin` is to centralize configurable behavior of our reinsurance implementation.

In the base configuration, PolicyCenter calculates the values for **Gross Retention**, **Ceded Risk**, **Target Max Retention**, and **Inclusion** that appear on the **Per Risk** tab in the policy file. Depending upon the characteristics of an individual risk, the insurer may want to override the default behavior. You can override the default behavior in the reinsurance configuration plugin. In the plugin, you can also specify the time component of the effective date for a new reinsurance agreement or program.

If you are integrating your own reinsurance system, you must create your own implementation of the methods and properties in the `IReinsuranceConfigPlugin` interface.

Getting default gross retention amounts for reinsurable risks

PolicyCenter calls the `getDefaultValue` method on `IReinsuranceConfigPlugin` to calculate the default value for the gross retention of a given `RIRisk`. PolicyCenter calls this method when a program is set on a `RIRisk`.

```
function getDefaultGrossRetention(ririsk : RIRisk) : MonetaryAmount
```

The method returns a `MonetaryAmount` with the calculated gross retention amount.

Getting inclusion types for risks and related reinsurance agreements

PolicyCenter calls the `getInclusionType` method on `IReinsuranceConfigPlugin` to obtain the default inclusion type for a given reinsurance agreement and `RIRisk`. PolicyCenter stores the inclusion type of an `RIAgreement` and `RIRisk` pair only if the inclusion type differs from the default value this method returns.

```
function getInclusionType(ririsk : RIRisk, agreement : RIAgreement) : RIAttachmentInclusionType
```

The method returns the default inclusion type for the given reinsurance agreement and `RIRisk`.

If you change this method to return different default values, this default value is changed, all policy periods that use earlier default values must change. The impact is large. It includes draft and bound branches, which in turn requires recalculating ceded premium amounts on affected policy periods.

Getting override ceded amounts for surplus reinsurance treaties

PolicyCenter calls the `getOverrideCededAmountForSurplusRITreaty` method on `IReinsuranceConfigPlugin` to set the ceded amount for a surplus treaty to an amount less than the maximum allowed by lines multiplied by gross retention. If the return value is not null, the PolicyCenter overrides the ceded amount. If the return value is greater than the maximum allowed amount, PolicyCenter uses the maximum allowed amount.

```
function getOverrideCededAmountForSurplusRITreaty(ririsk : RIRisk, agreement : SurplusRITreaty) : MonetaryAmount
```

The method returns the override amount to cede as a `MonetaryAmount`. If the method calculates no amount, it returns `null`.

The default implementation of the `getOverrideCededAmountForSurplusRITreaty` method does not calculate an override amount for how much to ceded on a surplus reinsurance treaty. It always returns `null` instead of a `BigDecimal`.

Effective time for new reinsurance programs and agreements

PolicyCenter gets the `ReinsuranceEffectiveTime` property to set the time component of the effective dates for a new reinsurance agreement or program.

```
property get ReinsuranceEffectiveTime() : Date
```

The method returns a `Date` with the time portion specified. Use the value returned as the time portion of reinsurance effective dates and intervals.

The base implementation of the `ReinsuranceEffectiveTime` property provides a constant `Date` value, `12:01 am`.

```
return "12:01 am" as Date
```

You might change this implementation if you want reinsurance effective time to begin at noon instead of at midnight.

Generating risk numbers for new reinsurable risks

PolicyCenter calls the `generateRiskNumber` method on `IReinsuranceConfigPlugin` to generate the unique identifier for a new reinsurable risk. The following reinsurance entities related to the reinsurable risk receive the same risk number that your plugin implementation returns.

- `Reinsurable` – A descendant type, such as a `PolicyRisk`, that represents the reinsurable risk
- `RiskVersionList` – List of `RIRisk` instances for the reinsurable risk
- `RIRisk` – Details of a reinsurable risk for a specific effective interval during a specific policy period

```
function generateRiskNumber() : java.lang.String
```

The method returns a `java.lang.String` with the unique number of a reinsurable risk.

The base implementation of the `generateRiskNumber` method uses the `gw.api.database.SequenceUtil` class to obtain a unique risk number, with the following, single Gosu statement.

```
return SequenceUtil.next(1, "RI_RISK_NUMBER") as String
```

You might replace the preceding statement with a call to your own reinsurance system to obtain risk numbers for new reinsurable risks stored in PolicyCenter.

Getting the targeted maximum retention for a reinsurable risk

PolicyCenter calls the `getTargetMaxRetention` method on `IReinsuranceConfigPlugin` to obtain the target net retention for a given `RIRisk`.

```
function getTargetMaxRetention(ririsk : RIRisk) : MonetaryAmount
```

This method returns a `MonetaryAmount` with the target net retention for a given `RIRisk`.

The default implementation of this method determines the targeted maximum retentions based on the reinsurance program associated with the `RIRisk`.

Whether to generate reinsurable risks on policy periods

PolicyCenter calls the `shouldPolicyTermGenerateReinsurables` method on `IReinsuranceConfigPlugin` to determine if reinsurable risks on a policy need to be generated for a new policy period.

```
function shouldPolicyTermGenerateReinsurables(period : PolicyPeriod) : boolean
```

The default implementation of the `shouldPolicyTermGenerateReinsurables` method always returns `true`. So by default, PolicyCenter always creates reinsurable risks on policy terms. If you are phasing in support of reinsurance, you might return `true` for policy periods after a specific date.

Whether a program covers a reinsurable

PolicyCenter calls the `programCanCoverReinsurable` plugin method on `IReinsuranceConfigPlugin` to determine if a program covers a given reinsurable. You can assume that the program is already the correct `RICoverageGroup` type.

```
override function programCanCoverReinsurable(program : RIProgram, reinsurable : Reinsurable) : boolean
```

You can use this method to specify additional risk criteria for the program. For example, you can specify whether programs cover particular regions.

The default implementation does not examine the reinsurable. It considers all programs applicable.

The `program` parameter is the program in that the method evaluates. The method attempts to apply the program to the reinsurable passed in the `reinsurable` parameter. This method returns `true` if the program can cover the reinsurable.

Choosing reinsurance programs for new reinsurable risks

PolicyCenter calls the `chooseReinsuranceProgram` method on `IReinsuranceConfigPlugin` to filter a list of reinsurance programs that might be applicable to a specific reinsurance coverage group on a specific effective date. PolicyCenter calls this method to choose the correct program from among the candidates.

If necessary, PolicyCenter attempts to assign the best program by first considering active programs, then draft programs, and finally programs from prior years. To search for the best program for the reinsurable risk, PolicyCenter calls this method up to three times, once for each search type. In the base configuration, the following method matches the currency of the reinsurable with the currency of the reinsurance program.

```
function chooseReinsuranceProgram(candidates : RIProgram[], reinsurable : Reinsurable,
                                   date : Date, searchType : SearchType) : RIProgram
```

The parameters are described below.

candidates

A list of candidate reinsurance programs with the correct `RICoverageGroup` type.

reinsurable

The reinsurable risk.

date

The effective date for which an applicable program is to be chosen.

searchType

The stage of the search by PolicyCenter for the best program for the reinsurable. The supported values are listed below.

- ACTIVE_PROGRAMS
- DRAFT_PROGRAMS
- PRIOR_YEAR_PROGRAMS

The method returns an `RIProgram` instance that represents a chosen reinsurance program.

Reinsurance ceding plugin

PolicyCenter provides the `IReinsuranceCedingPlugin` to help you integrate Guidewire Reinsurance Management and to let you enhance it with your own reinsurance ceding logic. You can find the registry for the `IReinsuranceCedingPlugin` in Guidewire Studio by navigating in the **Project** window to **configuration→config→Plugins→registry**. The source code is in the `IReinsuranceCedingPlugin.gwp` file.

Note: Guidewire Reinsurance Management is available within Guidewire PolicyCenter. To determine whether your Guidewire PolicyCenter license agreement includes Reinsurance Management, contact your Guidewire sales representative. Reinsurance Management requires an additional license key. For instructions on obtaining and installing this key, contact your Guidewire support representative.

Reinsurance ceding plugin implementations

PolicyCenter provides `PCReinsuranceCedingPlugin.gs` as an implementation of `IReinsuranceCedingPlugin`. It helps integrate Guidewire Reinsurance Management with PolicyCenter. To configure Guidewire Reinsurance Management with your own reinsurance ceding logic, modify `PCReinsuranceCedingPlugin.gs`. In Studio, you can modify the code directly, or you can define a subclass of `PCReinsuranceCedingPlugin` to override or extend specific methods.

In the base configuration, PolicyCenter calculates the ceded premiums and commissions and stores the values in a database table. You can integrate with an accounts payable system that processes the ceded premiums and commissions.

Calculations for proportional and facultative agreements only

The `PCReinsuranceCedingPlugin.gs` calculates ceded premiums for proportional and facultative agreements only. The implementation of the plugin does not calculate ceded premiums for non-proportional treaties. However, you can add such a calculation to the plugin.

See also

- For an example of how PolicyCenter calculates the ceded premium, see the *Application Guide*.

Reinsurance ceding plugin methods

The main purpose of `IReinsuranceCedingPlugin` is to calculate ceded premiums on covered reinsurable risks. All ceding calculations are done through the `PremiumCeding` work queue. To trigger a calculation, the entity that needs ceding calculations performed must be added to the queue. For example, if a policy period undergoes a branch promotion, its reinsurance ceding amounts must be calculated.

If you are integrating your own reinsurance system, you must create your own implementation of the methods in the `IReinsuranceCedingPlugin`.

Enqueuing policy periods for ceding calculations

The `PremiumCeding` work queue calls the `enqueueForCeding` method on `IReinsuranceCedingPlugin` to add a policy period to the work queue to calculate its ceding amounts. The work queue calls this method only for an initial ceding, not a full recalculation.

```
function enqueueForCeding(period : PolicyPeriod, reason : RIRecalcReason, comment : String)
```

The parameters are described below.

Parameter	Description
period	The policy period for which ceding calculations are needed.
reason	The reason for needing to recalculate ceding amounts, as a code from the <code>RIRecalcReason</code> typelist.
comment	An optional comment describing why a recalculation is requested.

To trigger calculations, an entity that needs ceding calculations must be added to the work queue. For example, with a branch promotion, the `PolicyPeriod` is added to the work queue.

Calculating ceded premiums for policy periods

The PremiumCeding work queue calls the `calculateCedingForPeriod` method on `IReinsuranceCedingPlugin` to calculate the ceding amounts for a given policy period.

```
function calculateCedingForPeriod(period : PolicyPeriod, recalculateAll : boolean,
    reason : RIRcalcReason, comment : String, updateUser : User)
```

The parameters are described in the following table.

Parameter	Description
period	The policy period for which ceding calculations are needed.
recalculateAll	Whether all ceding amounts on the policy period must be recalculated. If the value is true, the method must calculate all ceding amounts attached to the policy period. If the value is false, the method must calculate ceding amounts only for new reinsurable risks on the policy period.
reason	The reason for needing to recalculate ceding amounts, as a code from the <code>RIRcalcReason</code> typelist.
comment	An optional comment describing why a recalculation is requested.
updateUser	The user to assign the activity to if the work queue change caused an error in an <code>RIRisk</code> .

Whether to recalculate ceding amounts

The PremiumCeding work queue calls the `shouldRecalculateCeding` method on `IReinsuranceCedingPlugin` to determine whether to recalculate ceding amounts for a given work item.

```
function shouldRecalculateCeding(workItem : RICedingWorkItem) : boolean
```

The method takes a single parameter of type `RICedingWorkItem`, which is the item to evaluate for recalculation.

The method returns a `boolean` in which the value `true` indicates that ceding amounts must be recalculated on the work item.

Which user is responsible for reinsurance program changes

The PremiumCeding work queue calls the `userResponsibleForProgramChange` method on `IReinsuranceCedingPlugin` to determine which user is responsible for fixing validation errors caused by a program change.

```
function userResponsibleForProgramChange(period : PolicyPeriod, dirtyPrograms : RIProgram[]) : User
```

The parameters are described in the following table.

Parameter	Description
period	The period that the program affected.
dirtyPrograms	The list of programs that changed.

The method returns the user who is responsible for correcting validation errors. After calling this method, PolicyCenter assigns the user an activity to look at the job affected by the program change.

Logging errors for invalid reinsurance programs

The PremiumCeding work queue calls the `logErrorForInvalidPrograms` method on `IReinsuranceCedingPlugin` to log an error message for each invalid program added to the work queue for recalculating ceded premiums.

```
function logErrorForInvalidPrograms(programs : List<RIProgram>)
```

The method has a `programs` parameter which is a list of changed programs.

Reinsurance coverage web service

Use the RICoverageAPI web service to find reinsurable risk information stored in PolicyCenter. The web service is WS-I compliant.

PolicyCenter provides a reference implementation of the RICoverageAPI web service. From the **Resources** pane in Studio, navigate to **configuration**→**gsrc**→**gw**→**webservice**→**pc**→**pc1000**→**reinsurance**. Your claims system can use this web service to obtain reinsurance coverage information on policies involved in claims.

If you use ClaimCenter and PolicyCenter together, ClaimCenter uses this web service to synchronize reinsurance information for policies on claims. To integrate PolicyCenter with ClaimCenter for purposes of reinsurance, you must enable the **IReinsurancePlugin** in ClaimCenter and configure it to use the **PCReinsurancePlugin** implementation.

Methods of the PolicyCenter reinsurance coverage web service

The RICoverageAPI web service provides two methods so that external systems can find reinsurable risks in PolicyCenter.

- **findRIPolicyRisk** – Finds reinsurable risks of the specified reinsurance coverage group type that are attached to the policy on a particular date.
- **findRIRiskByCoverableID** – Same as above, but finds reinsurable risks only on the specified coverable in the policy.

Finding reinsurable risks by policy

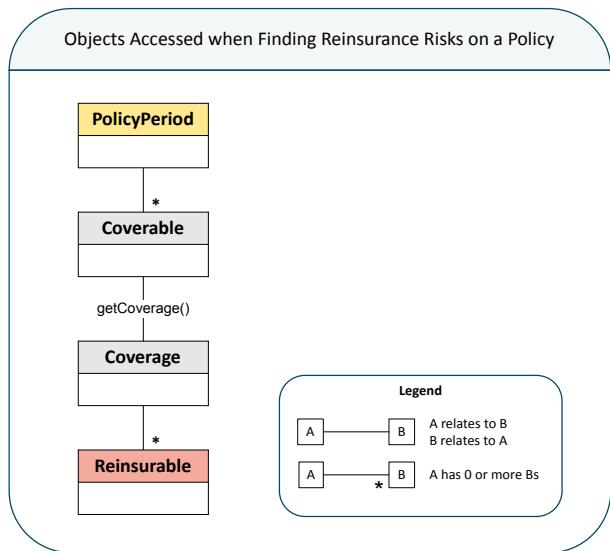
The **findRIPolicyRisk** method finds reinsurable risks of the specified reinsurance coverage group type that are attached to the policy on a particular date. A reinsurable risk exists on a policy. For example, the property coverages at a particular location constitute a reinsurable risk.

```
function findRIPolicyRisk(policyNumber : String, coverageCode : String, date : Date) : RIRiskInfo
```

The parameters are described in the following table.

Parameter	Description
policyNumber	A policy number of the policy to find risks in.
coverageCode	A code for the coverage pattern of a reinsurance coverage group type. The method finds risks of this reinsurance coverage group type.
date	A date on which to find these risks on the policy.

The following illustration shows some of the objects accessed by this method.



The method returns risk information in an `RIRiskInfo` instance.

See also

- “Reinsurance risk information” on page 447

Finding reinsurable risks by coverable on policy

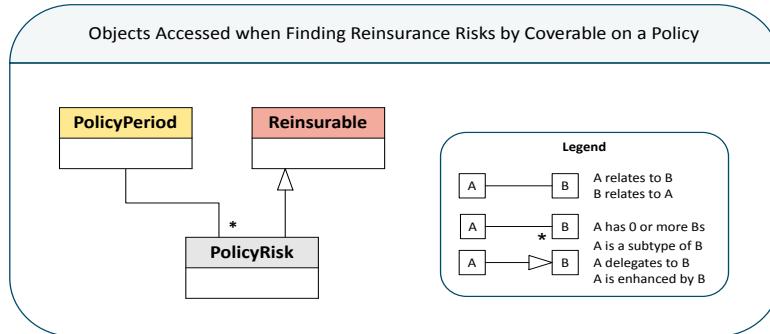
The `findRIRiskByCoverableID` method finds reinsurable risks of the specified reinsurance coverage group type that are attached to a specified coverable on the policy on a particular date.

```
function findRIRiskByCoverableID(policyNumber : String, coverableID : String,
coverageCode : String, date : Date) : RIRiskInfo
```

The parameters are described in the following table.

Parameter	Description
policyNumber	A policy number of the policy to find risks in.
coverableID	A public ID of the coverable in the policy to find risks on.
coverageCode	A code for the coverage pattern of a reinsurance coverage group type. The method finds risks of this reinsurance coverage group type.
date	A date on which to find these risks on the policy.

The following illustration shows some of the objects accessed by this method.

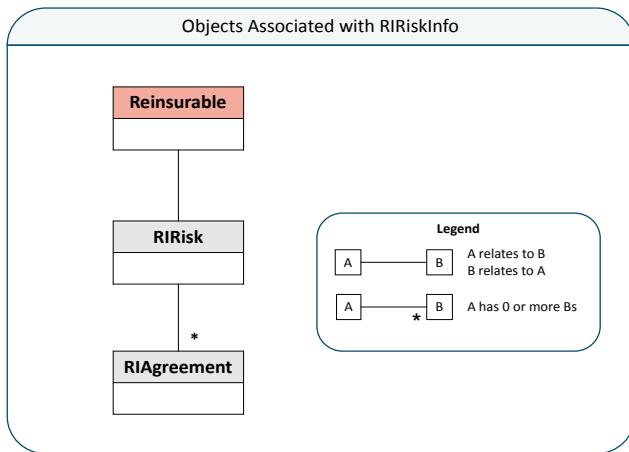


The method returns risk information in an `RIRiskInfo` instance.

Reinsurance risk information

Both the `findRIPolicyRisk` and `findRIPolicyRiskByCoverableID` methods of the `RICoverageAPI` web service return risk information in an `RIRiskInfo` data transfer object. In the base configuration, these methods obtain the `RIRiskInfo` by calling the `findReinsuranceRiskInfo` method on the `IReinsurancePlugin` plugin.

The following illustration shows some of the objects associated with the `RIRiskInfo` object.



RIRiskInfo object fields

The `RIRiskInfo` object has the following fields.

Field	Description
Agreements	An array of reinsurance agreement RIAGreementInfo objects, as described below.
Description	A description such as "Property coverage for Location 1" or "Auto liability coverage for Personal Auto line." The pattern for the description is shown below. <code>CoveragePattern.Name coverage for Reinsurable.DisplayName</code>
RIRiskID	The PublicID of the RIRisk attached to the Reinsurable.

RIAgreementInfo object fields

The **RIRiskInfo** object contains information about the reinsurance agreements. The object contains no agreement that is specifically excluded nor any proportional agreement with a 0% share. The object includes all non-proportional agreements. This object does not contain agreements in draft status.

The **RIRiskInfo** object contains the following information for each **RIAgreement** as an **RIAgreementInfo** data transfer object.

Field	Description
AgreementNumber	The AgreementNumber of the agreement.
AttachmentPoint	For non-proportional agreements only, the AttachmentPoint of the agreement.
AttachmentPointIndexed	For non-proportional agreements only, the AttachmentIndexed property of the agreement.
CededShare	The CededShare percentage of the agreement.
Comments	The Comments of the agreement.

Field	Description
Currency	The Currency of the agreement.
Draft	For treaties, true if the reinsurance program that contains the agreement is not yet active. For facultative agreements, true if the facultative agreement is not yet active.
EffectiveDate	The effective date of the agreement.
ExpirationDate	The expiration date of the agreement.
Name	The Name of the agreement.
NotificationThreshold	For per risk agreements only, the NotificationThreshold.
ProportionalPercentage	For proportional agreements, the proportional percentage of the risk that the attachment takes. In PolicyCenter, this value is the Prop % column for an agreement.
RecoveryLimit	For proportional agreements, the CededRisk of the attachment. For non-proportional agreements, the amount of reinsurance specified by the agreement as a calculated percentage.
TopOfLayer	For non-proportional agreements only, the CoverageLimit.
TopOfLayerIndexed	For non-proportional agreements only, the LimitIndexed property of the agreement.
Type	The Subtype of the agreement, such as quota share or surplus.

Forms integration

For an insured customer, the physical representation of an insurance policy is a collection of *forms*. Different forms define different aspects of a policy, such as coverages, exclusions, government regulations, and similar items. PolicyCenter supports forms in the user interface. This topic describes how to integrate an external forms printing service with PolicyCenter.

Although forms has a user interface for previewing what forms to create, the forms feature exists primarily to send data to an external forms printing system during issuance. Forms can also print for other jobs such as a policy change job that triggers reprinting a changed form, or printing additional forms that are now necessary.

You must perform the following tasks to integrate an external forms printing service with PolicyCenter forms.

- Create forms inference classes to generate XML – You must define Gosu classes that extend the `FormData` abstract class. The `FormData` class generates XML that describes data on the form that changes with each issuance of the form. The generated XML does not include introductory or boilerplate text that does not change.
- Write Event Fired rules that generate messages to a forms printing service – Event Fired rules intercept forms issuance events and generate messages to the external forms printing service. The message payload might be simply the XML generated by the forms inference classes.
- Write forms messaging code to talk to the external system – Your custom messaging plugins (and new messaging destinations) that you register must send the XML payload to the forms printing system.

[See also](#)

- [Application Guide](#)

Forms inference classes

You configure basic forms settings in Product Designer by creating and editing a `FormPattern` in the **Forms** tab in a policy line window. As part of that editor, you provide the fully-qualified class name of a Gosu class that extends the abstract class `FormData`. Your class must generate data that describes the variable data of the form. This data must not include boilerplate text. Such classes are known as forms inference classes.

You can subclass `FormData` directly for each form, or create your own subclasses of `FormData` that provide common behaviors or data. For example, PolicyCenter includes a class called `PAFormData` that defines some common behaviors for the built-in personal auto forms. Individual forms extend `PAFormData` instead of `FormData`.

After a job finishes, PolicyCenter triggers forms inference and then raises a messaging event. You can catch the messaging event in your Event Fired Rules to generate messages to the external forms printing system. For example, the submission job process contains code like that shown below.

```
FormInferenceEngine.Instance.inferPreBindForms(_branch)
_branch.addEvent("IssueSubmission")
```

Creating inference data and XML

There are two main steps to creating the inference data in your subclass of `FormData`.

- Create inference data, which can be any Gosu data that you store in private properties in the `FormData` instance. This is used later by other methods in your inference class.
- Create XML data from inference data using the inference data that your class generated previously. Use `XmlNode` objects in Gosu to generate the XML-formatted data.

Inference data

The important method for creating inference data is the abstract method `populateInferenceData`. It is called by the forms inference engine to populate this instance with the appropriate data from the policy graph. The method is called immediately after the instance of the `FormData` is created.

```
abstract function populateInferenceData(context : FormInferenceContext, availableStates : Set<Jurisdiction>)
```

The `context` argument contains information, such as the `PolicyPeriod`, the set of forms in the group, and the differences between periods.

The `availableStates` argument contains all the states in which the form was found to be available. Any state-specific form that replaces the national version is not included in the set.

The recommended approach to implement the method is to generate and then store inference data in a private variable so it can be read by other methods in your class. The `Form_Example` class shown below assumes a built-in personal auto form that can create towing labor coverage data like that referenced in the sample `populateInferenceData` method.

```
@Export
class Form_Example extends FormData {

    var _towingInfo : List<towingInfoSet>

    override function populateInferenceData(context : FormInferenceContext, availableStates : Set<Jurisdiction>) {
        var towingInfoSet = mapVehicles(context, \v -> v.PATowingLaborCovExists, \v -> createTowingInfo(v))
        _towingInfo = towingInfoSet.toList().sortBy(\info -> info.Vin)
    }
}
```

In the example, `mapVehicles` uses a `mapVehicles` method implemented in `PAFormData`. It uses the helper method `mapArrayToSet`. The result of `mapVehicles` is a Map object mapping vehicles to towing information data created by another helper method. Finally, the result is converted to a `java.util.List` object and sorted by the vehicle identification number. This result must be stored in a private variable defined in your `FormData` subclass. Note that the final result is a `List` and not an `XmlNode` or XML-formatted text.

The subclassed `FormData` class must also implement the abstract getter method `InferredByCurrentData`. The method indicates whether the form is part of the policy.

```
abstract property get InferredByCurrentData() : boolean
```

The method returns `true` if the form is part of the policy. Even if the method returns `true`, however, does not guarantee that the form will be added to the policy. The final determination depends on the processing type specified in the `FormPattern` and whether the data on the form matches the data on any previous version of the form.

The following code continues implementing the sample `Form_Example` class. The implemented `InferredByCurrentData` method references the class's `_towingInfo` list initialized in the sample `populateInferenceData`. The method returns `true` if the list contains any data.

```
override property get InferredByCurrentData() : boolean {
    return !_towingInfo.Empty
}
```

XML of inference data

The important method for exporting the inference data is the abstract method `addDataForComparisonOrExport`.

```
abstract function addDataForComparisonOrExport(contentNode : XMLNode)
```

The `contentNode` argument contains an XML node to add data to.

The method has no return value.

The implemented method must take the inference data stored in private variables and add child XML nodes to the `contentNode` object. The resulting node will be used to compare two forms to discover any changes that may have occurred, such as if a form uses the "reissued" policy form. Any discovered changes will be persisted to the database.

The sample `Form_Example` class could use the following code to generate XML data from the `_towingInfo` list variable.

```
override function addDataForComparisonOrExport(contentNode: XMLNode) {
    var node = createScheduleNode("Vehicles",
        "Vehicle",
        _towingInfo.map(\info -> info.Vin + " - " + info.Premium))
    contentNode.Children.add(node)
}
```

The XML output of your inference exports to text and persists to the database with the form. This persisted version of the XML data has a special purpose. It determines whether a change to a policy triggers printing of a new form.

The XML data contains only the data that changes and is unique to this policy on the form. If the policy changes but your forms inference class generates the same XML for it, by definition the form did not change. If the XML exported is different after a policy change or if it is newly available, PolicyCenter knows that this is a new form.

In addition to the `PolicyPeriod.Forms` property that contains all forms for the policy, PolicyCenter tracks new forms specially from the complete list of forms in the `PolicyPeriod.NewlyAddedForms` property. Some of these forms may be new forms and some may be forms to reprint because of recent policy changes.

You might need one form to be duplicated for a series of items, such as separate duplicate forms for each vehicle rather than one form that lists all vehicles.

Because PolicyCenter uses the XML output to determine whether forms need to be reprinted, in general your XML contains only the critical variable data for this form. Be careful not to include extra data that might falsely tell PolicyCenter that this form must be reprinted. However, in some cases you might want to include XML data that might be useful metadata for your message to the external system but not to compare forms for changes.

You can omit certain nodes or information in the node by adding special attributes to the XML nodes. Additionally, the `FormData` data class has methods on it you can use to conveniently add these attributes to an existing `XMLNode`. If you call the methods, they return the original node back to make it easier to wrap or chain method calls to multiple methods or other APIs.

For example, you can store data in attributes and use important codes or IDs for comparisons. You can, however, ignore attributes such as class code descriptions that can change without requiring forms to be reprinted. Use the `ignoreAttributes` attribute for this feature.

Alternatively, if you package your data as text content on a child node, set child nodes for package names to `ignoreAll` so PolicyCenter ignores them during comparison.

The following table lists the purpose, the attribute on an `XMLNode` that you can set to have this behavior, and the method name you can use to add this attribute.

Attribute name and method name	Description
<code>ignoreAll</code>	Indicates a node to ignore during comparison. If comparing children of two XML nodes, PolicyCenter strips out and ignores all child nodes if its <code>ignoreAll</code> attribute has the value <code>true</code> .
<code>ignoreAllAttributes</code>	Indicates to ignore all attributes on a node during comparison. Set this attribute value to <code>true</code> for this behavior.

Attribute name and method name	Description
ignoreAttributes	Indicates that a node with a list of attributes to ignore during comparison. For the method version of <code>ignoreAttributes</code> , the attributes are an array of <code>String</code> values. For the attribute <code>ignoreAttributes</code> directly on the <code>XMLNode</code> , the attribute value must be a comma-delimited list of attribute names. You must not include space characters between the values before nor after the commas.
ignoreChildren	Indicates to ignore the children of a node during comparison. Set this attribute value to <code>true</code> for this behavior.
ignoreText	Indicates to ignore the text of a node during comparison. Set this attribute value to <code>true</code> for this behavior.

The most important attributes and methods are `ignoreAll` and `ignoreAttributes`. The other methods exist primarily for completeness.

Determining whether to add or reprint a form

The important method for creating inference data is the `FormData` property accessor function `ShouldBeAdded`. The application calls this only after your `FormData` object creates inference data. It must return `true` if the form is part of the policy, or `false` if the form is irrelevant to the policy with the current policy data. Returning `true` from this method does not guarantee PolicyCenter adds the form. This is affected by other properties such as the processing type specified in the `FormPattern` in Product Designer, the group code, and whether the XML data changes and triggers reprinting.

PolicyCenter only calls this after your `FormData` object creates inference data. A simple way of implementing this method is to check whether your inference data in your private variable is non-empty.

As an example, the inference class described earlier in this section could use the code shown below.

```
override property get ShouldBeAdded() : boolean {
    return !_towingInfo.Empty
}
```

This method returns `true` only if the private variable for inference data contains a non-empty list.

Form data helper functions

The `FormData` class that you extend your class from includes some helper methods for common tasks.

Method	Description
<code>createScheduleNode</code>	Creates a parent XML node and a list of child nodes. You specify a container name, a child element name, and a list of children elements. Specify the list of children elements as an iterable collection of <code>String</code> values for the text value for each child node.
<code>createTextNode</code>	Creates an XML node with the specified name and text content.
<code>mapArrayToSet</code>	Given an array, a filter (a Gosu block) and a mapping operation (a Gosu block), this method produces a set as the output. The set contains the result of the mapping applied to every element in the array for which filter returns true. If that filter argument is <code>null</code> instead of a block, PolicyCenter processes all elements. If you pass an array workers' compensation exposures, you can use this method to produce a set of states that have exposure for a given class code. To do this, use the filter argument to accept only exposures with the given class code and write a mapping block to extract the state from the exposure.

Additionally, several useful generic inference classes are defined in the package `gw.forms.generic` that might help in your form design. The following table lists helper classes in this package.

Class	Description
<code>AbstractMultipleCopiesForm</code>	Abstract class that you can subclass to easily deal with forms with multiple instances of the same form attached to the policy. This class assumes that the forms have a one-to-one relationship with some entity on the policy. The class also assumes there is a corresponding <code>FormAssociation</code> entity subtype that tracks which form points to which entity.
<code>AbstractSimpleAvailabilityForm</code>	Base class for any form that does not need any data to be gathered and packaged, but needs a simple availability script. Subclassing classes must implement the <code>isAvailable</code> method to indicate whether or not to add the form.
<code>GenericAlwaysAddedForm</code>	Base class for any form to always add to a policy whenever the form is available. Using this class with no further subclassing directly leads to a form with no data populated. However, you can extend this class and override the <code>addDataForComparisonOrExport</code> method to output your data.
<code>GenericRemovalAndReplacementEndorsementForm</code>	Base class for removal and replacement endorsement form. It checks if any forms were completely invalidated or replaced and that also set this form as their removal endorsement form number. If such a form is found, PolicyCenter creates a form that contains a parent <code>RemovedForms</code> node with a child <code>RemovedForm</code> node for each form completely invalidated. Underneath that node are nodes for <code>Description</code> , <code>EndorsementNumber</code> , and <code>FormNumber</code> nodes that describe the removed form. This class creates a comparable structure for forms replaced by a new copy of the same form.
<code>GenericRemovalEndorsementForm</code>	Base class for a generic removal endorsement form. It checks if any forms were completely invalidated and that also set this form as their removal endorsement form number. If such a form is found, it creates a form that contains a parent <code>RemovedForms</code> node with a child <code>RemovedForm</code> node for each form that became completely invalidated. Underneath each of those nodes are <code>Description</code> , <code>EndorsementNumber</code> , and <code>FormNumber</code> nodes with the actual data about the form that became invalidated.

The usage of the term “generic” in the package and class naming of the preceding classes means the classes include common functionality that you can subclass as desired. They do not necessarily use Gosu generics features. However, the `AbstractMultipleCopiesForm` class does use Gosu generics.

Handling multiple instances of one form

In many business cases, each form (as defined by a `FormPattern`) has either zero or one instances of the forms valid on the policy at any given time. For example, the personal auto line of business forms in the reference implementation include various forms, but no more than one of each form. For multiple vehicles, a single form lists all the vehicles. However, sometimes your business case might require multiple instances of the same form, such as listing one form multiple times for each vehicle. PolicyCenter supports this feature, which is referred to as multiplicity.

The most important part of handling multiplicity is associating a certain object in the policy graph with a given instance of your inference class, which is your `FormData` subclass. PolicyCenter implements this link through an intermediate business entity called `FormAssociation`. You must create a subtype of `FormAssociation` and add a single property on it that is a foreign key to the object that it represents. For example, for a personal auto form associated with single personal vehicle, create a new subtype of `FormAssociation` called `PAVehicleFormAssociation`. Add a property to it called `Vehicle` that links to the `PersonalVehicle` entity.

If you have multiple instances of the form, your `Form` has multiple instances, each with its own instance of your inference class.

Support form multiplicity

You use a subtype of `FormAssociation` and an inference class that extends `AbstractMultipleCopiesForm` to provide form multiplicity on a policy.

Procedure

1. In the data model, create a subtype of `FormAssociation`.

For a personal auto form associated with single personal vehicle, create a new subtype of `FormAssociation` called `PAVehicleFormAssociation`.

2. On that entity, add property with a foreign key link to the desired type, such as a personal vehicle.

Suppose the property is called `Vehicle`.

3. In your inference class, extend the `AbstractMultipleCopiesForm` class instead of the extending the `FormData` class. This class uses Gosu generics features to abstract this class for your linked-to type.

Use the following syntax to define the inference class.

```
class MyPAVehicleForm extends AbstractMultipleCopiesForm<PersonalVehicle>
```

4. To support form multiplicity, add the `getEntities` method to your inference class. This method must return a list containing one or more entities of your desired type. If there is more than one entity returned, PolicyCenter knows that there are multiple instances of the form, each linked to one of the items in this list.

`getEntities` returns a list of all personal vehicles on the policy.

Do not implement the `populateInferenceData` method. That method is used only if you do not use form multiplicity.

5. Add to your inference class the `FormAssociationPropertyName` method, which returns the name of the property in your `FormAssociation` subtype, as a `String`.

Pass the `String` value "Vehicle" because the property name is `Vehicle`.

6. In your inference class XML generation method `addDataForComparisonOrExport`, export the data for only one entity in your generated list of entities. Your code refers to the current entity by using the private variable called `_entity`, which is set up by the `AbstractMultipleCopiesForm`.

The `_entity` variable contains one `PersonalVehicle` from the list returned in your `getEntities` method. Your `addDataForComparisonOrExport` would add child XML nodes for that vehicle only.

7. In your inference class, create the `createFormAssociation` method to instantiate one version of your form association subtype.

This method can use simple code, such as the following example.

```
override protected function createFormAssociation( form : Form ) : FormAssociation {
    return new PAVehicleFormAssociation(form.Branch)
}
```

Next steps

In your messaging code, you must access `Form.FormAssociations` to get form associations that link to the associated entity.

Advanced multiplicity with multiple form associations per form

In the earlier multiplicity example, one `FormPattern` created multiple form instances each with one inference class instance and a single form association. The form was duplicated for each entity returned from `getEntities`, but each form linked to only one vehicle. This approach works for most cases in which you might need multiplicity on a form.

In the data model, the `Form` entity supports multiple form associations by having form associations on the `Form` entity stored in an array (not a single property) called `FormAssociations`. However, in the reference implementation the `FormData` and `AbstractMultipleCopiesForm` classes support only a one-to-one mapping of `Form` instance to `FormAssociation`. It is possible to support one-to-many mappings of `Form` to `FormAssociation`.

Supporting one-to-many mappings requires subclassing the built-in Gosu classes to manage multiple `FormAssociation` objects. Your subclasses set their values and set the `_entity` variable before calling the XML export method. Refer to the source code for the `AbstractMultipleCopiesForm` class to see how the multiplicity code is implemented.

Reference dates on a form

There is now a protected method in `FormData` called `getLookupDate` that determines the reference date to use for that form. The built-in class defines the standard implementation.

Individual form patterns can add their own implementation of the `getLookupDate` method in their forms inference class. For example, if a particular form is directly related to a particular coverage, the `getLookupDate` implementation for that form must return the reference date from the related coverage. In other words, you do not need to necessarily use the generic period-level reference date.

The built-in `FormData.getLookupDate` implementation gets the lookup date from coverages, which calls the `IReferenceDatePlugin` plugin implementation to get reference dates. However, `IReferenceDatePlugin` does not do anything specific to forms.

Forms messaging

The role of your messaging code for forms integration is to examine the list of newly added forms in the `PolicyPeriod.NewlyAddedForms` property and generate messages for the newly added forms.

Your messaging destination that represents your forms printing system must listen for the issuance events such as `IssueSubmission`, `IssuePolicyChange`, `IssueReinstatement`, and others for other jobs.

Your Event Fired rules must capture these events and generate one message for each form, or one message that contains the data for all the newly added forms.

Within each form there is a property that contains the cached XML data created by the inference class. You can get the cached XML data using code like that shown below.

```
cachedXML = myForm.FormTextData.TextData
```

You might make your message payload simply the XML payload generated from that `XmlNode` such as the following code.

```
var msg = MessageContext.createMessage(cachedXML)
```

If you support form multiplicity, remember that you might need to access the `Form.FormAssociations` property to access the form associations to access the linked entity to generate the payload.

Messaging plugins

Design your messaging transport plugin to send this message to the external forms printing system. Depending on the external system, you could send the XML directly to the system (or to integration code that manages the system).

Alternatively, if your message contained XML, you could traverse the XML data and format the data as appropriate for your printing system. If you use this approach, use XML APIs to convert the XML `String` data to `XmlNode` objects.

Creating documents

If your forms printing system can integrate with a document management system (DMS), the printing system can generate a visual representation of the form and add it to the DMS. After it does this, the integration code can call back into PolicyCenter to let it know there is a new document associated with the policy.

See also

- “Document management” on page 195

Policy difference and comparison customization

Comparing and copying policies and other entity instances in the policy graph are important tasks in policy management. If PolicyCenter cannot determine correctly whether two items in the database are actually the same item in the real world, you risk creating duplicate items or merging two different items. If you create new entity types that relate to real-world items, Guidewire recommends that you develop classes to compare and copy instances of these types. These classes are called matcher and copier classes respectively. You must test these classes rigorously during your configuration of PolicyCenter so that you can resolve problems in your implementations before PolicyCenter is in production. Similarly, if you make changes to entity types that exist in the base configuration of PolicyCenter, you must review the matcher and copier classes for those entity types. If necessary, you can extend the existing matcher and copier classes so that they include your new fields or behavior.

PolicyCenter compares two policy branches for many reasons. A branch is a snapshot of a policy at a specific model time for a specific logical policy period. For example, PolicyCenter compares two branches to show what changed in a policy change job or to compare differences in a multiple-version quote. You can modify the underlying logic that determines the differences between two policy branches. You can also change the appearance of policy differences in the PolicyCenter user interface.

The base configuration of PolicyCenter includes a reference implementation of the policy difference plugin in `gw.plugin.diff.impl.PolicyPeriodDiffPlugin`. You modify this class or implement your own class to customize the comparison of policy branches.

PolicyCenter provides API classes for comparison of policy branches in the `gw.api.diff` package and implementations of abstract classes that you can customize in the `gw.api.diff.impl` package.

Overview of policy differences

You can view comparisons between policy periods in several places in the PolicyCenter user interface. You can customize the appearance of these differences and customize the underlying logic that determines the differences.

The policy review screen that you can view as part of a policy change job is most common user interface for viewing policy differences.

The top-level trigger for most of the policy-difference code is PolicyCenter application logic calling the registered plugin for the `IPolicyPeriodDiffPlugin` interface. The plugin has the methods `filterDiffItems` and `compareBranches`.

Each difference that the plugin method generates is encapsulated in a difference item (`gw.api.diff.DiffItem`) object. The difference item represents a single difference between the policy periods. A difference has one of the following types:

- An addition of a bean (entity instance) to the policy period graph
- A removal of a bean from the policy period graph
- A change in a property
- A change in the policy period range

Each type of difference is implemented by a separate subclass of `DiffItem`.

The plugin methods implement different strategies to generate and filter the list of differences between two policy periods. The application flow determines which of these methods is called by user interface.

Difference item generation strategy	Description	What generates differences?	What filters differences?
Database generation of difference items	<p>PolicyCenter compares a policy period to the revision that it was based on.</p> <p>PolicyCenter can calculate changed properties and entities directly from the database. This process is very efficient and accurate. PolicyCenter uses this approach for policy changes and some other application flows.</p> <p>Your code filters out irrelevant differences.</p>	<p>Typically, PolicyCenter generates the differences by using a database query.</p> <p>A rare exception is for Workers' Compensation and General Liability. In this case, the database query does not generate exposure splits, so the <code>filterDiffItems</code> method in the plugin generates additional differences. See the <code>WCDiffHelper</code> and <code>GLDiffHelper</code> classes. These classes handle exposures that split.</p>	<p>The entry point for filtering is the <code>IPolicyPeriodDiffPlugin</code> plugin method <code>filterDiffItems</code>.</p> <p>Only customize that method to filter out differences if you are sure no PolicyCenter code ever needs those differences. For example, the application uses the list of differences in the following places.</p> <ul style="list-style-type: none"> • Logic that merges or applies changes from one revision to another. • User interface code to display differences. <p>You can filter dynamically based on the difference reason, by using the <code>DiffReason</code> typecode parameter on the plugin method.</p> <p>To configure the differences that the tree view displays and how the differences appear, customize a difference tree configuration XML file.</p>
Comparison of two branch graphs	<p>For multi-version jobs and in some other contexts, the policy period difference plugin triggers Gosu code that compares two arbitrary branch graphs and generates a list of differences.</p>	<p>The <code>compareBranches</code> method of the policy period difference plugin uses Gosu code to generate the differences.</p>	<p>This strategy is mostly about generating difference items rather than filtering them. This approach does use a secondary filtering process.</p> <p>To configure the differences that the tree view displays and how the differences appear, customize a difference tree configuration XML file.</p>

PolicyCenter chooses which strategy and, therefore, which plugin method to call based on the context of the request. The following table lists the places where PolicyCenter checks for differences between policy periods, how the check is implemented, and which strategy is used to generate the difference items.

Reason to check for differences	Strategy for generating differences	Description
Policy change transaction	Database generation	The Policy Review page displays what changed between the active branch and the branch upon which it was based.
Renewal transaction	Database generation	The Policy Review page displays what changed between the active branch and the branch upon which it was based.

Reason to check for differences	Strategy for generating differences	Description
Rewrite transaction	Database generation	The Policy Review page displays what changed between the active branch and the branch upon which it was based.
Out-of-sequence jobs (of multiple types)	Database generation	PolicyCenter must check whether a job is out of sequence and generate a list of conflicts, if any. If there are conflicts, users must review the list of changes to merge forward to future-effective-dated branches.
Preempted jobs	Database generation	<p>PolicyCenter uses differences in two places in the preemption flow.</p> <ul style="list-style-type: none"> First, PolicyCenter checks whether a job was preempted. If so, PolicyCenter displays and applies differences between the current branch and its based-on branch. Users must review the list of changes to apply from the preempting branch to the current branch before binding the active job. After PolicyCenter handles the preemption, some changes might conflict with changes already made in the preempted branch. PolicyCenter uses the differences engine and user interface to display those differences.
Integration	Database generation	Integration code might need to notify a downstream system of recent changes. A typical case is comparing a recently bound branch to the branch it was based on. You can generate changes, filter them for your downstream system, and then convert the results to a messaging payload.
Multi-version job policy comparisons	Compare two branch graphs	For a multi-version job, users can compare different versions of a policy and quotes for each one. The two or more non-bound branches share the same branchID but the branches are not based directly on each other. Some data is simplified or omitted for display. For example, for the Personal Auto line, the interface compares vehicles, coverage amounts, and resulting costs in the quote. If you add a vehicle, PolicyCenter adds coverages for the new vehicle but does not display the coverages explicitly in the comparison.
Arbitrary historical revision comparisons	Compare two branch graphs	To compare the state of the policy at different times in the policy history, users can select two versions in the same logical period. Even if two branches are in the same logical period, two arbitrary revisions a user wants to compare may not be directly based on each other. PolicyCenter cannot use the efficient database-based comparison mechanism in this case.

The full set of differences that PolicyCenter needs in order to perform the underlying data actions might be more than you want users to view in the user interface.

For example, some application tasks require generating multiple difference items, which you might consolidate and show as a single difference to the user. For another task, you might want to omit some differences between low level properties or window mode changes. You configure what difference items to display in the tree view within the user interface by using a difference tree configuration XML file. You must define one difference tree XML file for each product.

Difference item subclasses

PolicyCenter provides several types of difference items, all of which are subclasses of the `gw.api.diff.DiffItem` class. The reference implementation of the policy period difference plugin instantiates only property-change and bean-addition difference items. You can customize the reference implementation to use any of the classes, according to your business requirements. The following table describes these classes. The rightmost columns in the table indicate which objects are instantiated in the reference implementation in different contexts.

Class	Description	Created if comparing two branch graphs	Created by database generation of difference items
DiffAdd	An entity instance was added that did not exist previously.	Yes	Yes
DiffProperty	A property changed in an entity instance.	Yes	Yes
DiffRemove	An entity instance was removed that existed previously.	Yes	Yes
DiffWindow	A change in window mode to an entity instance's effective or expiration dates. Window mode means the change is made while viewing a <code>PolicyPeriod</code> and its subobjects across all effective dates in the period. This object is not created in a multiple revision job such as a multiple revision quote. PolicyCenter can add a <code>DiffWindow</code> object during comparison of two branch graphs. Comparing two branch graphs may pertain to multi-version or comparing jobs.	Yes, but not in multiple-revision jobs	Yes

You can create these difference items directly by using their constructors, with code like the following Java lines.

```
DiffAdd diff = new DiffAdd(newElem);
diff.setPath(ctx.getPath());
```

Alternatively, use APIs to create difference items by comparing entities in two branches.

Comparing branches by using difference helper classes

One of the two difference item strategies is to compare two arbitrary branches by using the policy period difference plugin `compareBranches` method.

In the reference implementation of this plugin, the `compareBranches` method compares policies by using helper classes that examine details for each line of business. The base configuration includes a separate difference helper class for each line of business. The class name of the helper class includes the line of business name.

For example, for each line of business, the reference implementation of the plugin calls the `createPolicyLineDiffHelper` method applicable to that policy line. The `createPolicyLineDiffHelper` method is responsible for returning the correct difference helper class, which is a subclass of `DiffHelper`. For example, for personal auto business line, the method instantiates an object of class `PADiffHelper`.

Next, the Gosu code calls the `addDiffItems` and `filterDiffItems` methods on that helper class.

```
// Add diffs by line of business
for (line1 in p1.Lines) {
    var line2 = p2.Lines.firstWhere(\ l -> l.Subtype == line1.Subtype)
    if (line2 != null) {
        diffHelper = line1.createPolicyLineDiffHelper(reason, line2)
        if (diffHelper != null) {
            diffItems = diffHelper.addDiffItems(diffItems) as ArrayList<DiffItem>
            diffItems = diffHelper.filterDiffItems(diffItems) as ArrayList<DiffItem>
        }
    }
}
```

If you make new lines of business in PolicyCenter, create new helper classes named with the line of business as a prefix. Your helper class encapsulates your difference logic for that line of business. Your helper class must extend the built-in `DiffHelper` class. Your subclass must encapsulate the line-specific logic and call methods on other classes such as `DiffUtils` as necessary.

Difference APIs

Two API classes are available for customizing the policy differences for a policy line: the `DiffHelper` class and the `DiffUtils` class. Use these classes primarily from the subclass of `DiffHelper` that is unique to that policy line. For example, for the personal auto line, use the `PADiffHelper` class.

To see examples of usage of both types of difference API, in Studio, look at `WCDiffHelper`.

Difference methods on the DiffHelper class

Each line of business has its own subclass of `DiffHelper`. For example, for personal auto, the subclass is the `PADiffHelper` class. The `PADiffHelper` and other subclasses of `DiffHelper`, provide utility methods that can help you.

`compareEffectiveDatedFields`

Compare a field on the effective dated fields to a configurable graph depth as an argument. Effective dated fields is an object on the policy period. This object contains various fields that need to be effective dated. Those fields cannot be on the `PolicyPeriod` because `PolicyPeriod` is always the root of the graph and never is duplicated across effective time.

`compareLineField`

Compare a field on the line (passed as a `String` of the property name) to a configurable graph depth as an argument.

`comparePolicyPeriodField`

Compare a field on the line to a configurable graph depth as an argument.

For example, the `compareLineField` method compares a `String` property to a specific depth in the second argument.

```
diffItems.addAll(this.compareLineField("Vehicles", 3))
```

For advanced comparisons, you can use the methods on the `DiffUtils` class by accessing the `DiffUtils` property in the `DiffHelper` base class. The property is an instance of the `DiffUtils` class that is instantiated with the correct bean matcher.

Difference methods on the DiffUtils class

`DiffUtils` is a Java class that compares two branches and helps filter a list of different items. You can use this class for advanced difference comparison. You access `DiffUtils` methods by using the `DiffUtils` property in the `DiffHelper` base class. That property is an instance of the `DiffUtils` object that is instantiated with the correct bean matcher. Each line of business has its own subclass of `DiffHelper`. For example, for personal auto, the subclass is the `PADiffHelper` class.

One of the important tasks of the `DiffUtils` instance is to compare two branch graphs and generate difference items. To customize this task, you specify the following information to an instance of the `DiffUtils` class.

- A bean matcher instance – During instantiation of `DiffUtils`, you pass a bean matcher instance in the constructor. A bean matcher is an object that compares two objects and determines if they are equal by looking at property data rather than at a `fixedID` or primary key. For example, you can determine whether two cars are the same by checking their vehicle identification number. You can determine whether two contacts (`PolicyContact` objects) are the same by checking the name and address. You do not have to write a custom bean matcher, although you can customize the existing logic. The bean matcher in the PolicyCenter base configuration handles objects in the PolicyCenter reference implementation of the policy difference plugin.
- Set of excluded types – Each `DiffUtils` instance includes a list of types to exclude. To add excluded types, call the `excludeType` method.
- Set of excluded properties – Each `DiffUtils` instance includes a list of properties to exclude. To add excluded properties, call the `excludeField` method. To see syntax and examples, in Studio view the usage in the `DiffHelper` class.

- Set of included properties – Each `DiffUtils` instance adds a list of properties to include, even if excluded by other rules. To add included properties, call the `includeField` method.

The typical methods that you use on the `DiffUtils` class are described below. To see examples of their usage, view the `DiffHelper` class.

- `compareBeans` – Compares two entity graphs. This method compares all properties and traverses all arrays and links to a specified depth, which is the number of steps down the hierarchy to examine.
- `compareField` – Compares two entity graphs starting at a specific property. The property can be a column, link, or array. From that property, `compareField` compares all properties and traverses all arrays and links to a specified depth, which is the number of steps down the hierarchy to examine. Pass the property as an `IEntityPropertyInfo` object, which you can get from the type system by using the following syntax.

```
PersonalAutoLine#PALineCov... PropertyInfo
```

The `#` syntax identifies a feature literal such as a property or method. The `compareField` method returns an `ArrayList` of differences as `DiffItem`.

To customize the built-in behavior, add logic to the class that instantiates `DiffUtils`. For example, for personal auto line of business, modify the `PADiffHelper` class, which extends `DiffHelper`, which instantiates `DiffUtils`. The `addDiffItems` method adds the differences.

The following example code instantiates the `DiffUtils` class and compares two business lines. The code adds differences for properties `PersonalAutoLine.PALineCovages` and `PersonalAutoLine.Vehicles`.

```
...
// Create a new instance of DiffUtils
var diffUtils = new DiffUtils(new PCBeanMatcher()) // or any other custom BeanMatcher implementation

// Include the policy lines properties
diffUtils.includeField(PolicyPeriod#Lines.PropertyInfo)

// Calculate (add) line coverage differences by comparing a field
diffItems.addAll(diffUtils.compareField( line1, line2,
    PersonalAutoLine#PALineCovages.PropertyInfo , 2))

// Calculate (add) vehicle differences
diffItems.addAll(diffUtils.compareField( line1, line2,
    PersonalAutoLine#Vehicles.PropertyInfo, 2))
...
```

Comparing individual objects by using matchers

A few parts of PolicyCenter need specialized logic to determine whether two objects represent exactly the same thing. For example:

- The policy difference page, such as for a policy change job, comparing a policy to the original. This page supports other comparisons, such as for side-by-side quoting for multiple draft revision jobs.
- Out-of-sequence job handling.
- Preemption job handling.

To determine whether two objects represent the same real-world item, PolicyCenter uses matcher classes. For example, to compare two cars, you might compare their unique vehicle identification numbers (VINs). If the VINs match, the objects represent the same car, regardless of other properties on the car objects that have different values.

The following types of classes for matching objects are available in PolicyCenter.

- Delegate-based matching classes
- `PCBeanMatcher`, which is an older type of matching class

The `PCBeanMatcher` class provides the base comparison of the `Id` property for all entity types. For effective-dated subobjects in a `PolicyPeriod` graph, the base comparison in the `PCBeanMatcher` class also compares the `FixedID` property of the objects. If the properties that the base comparison checks are the same for both objects, the `PCBeanMatcher` class performs additional tests. In most cases, a delegate-based matching class performs the additional tests. For a very few entity types, the `PCBeanMatcher` class performs the tests.

Planning how to match entity instances

Matching entity instances is an important task in setting up your PolicyCenter implementation. When you create a new entity type or extend an existing type, you must consider which matching strategy to implement. You must also test your matcher implementation rigorously to confirm that you do not have the following types of matching errors:

- Matching two entity instances that actually refer to different items. This error causes PolicyCenter to discard a valid object.
- Not matching two entity instances that actually refer to the same item. This error causes PolicyCenter to create an invalid duplicate object.

Matching two objects is straightforward for transactions that are in sequence because you can rely on the `ID` and, for effective-dated objects, the `FixedID` properties. Transactions that occur out of sequence require careful consideration.

Writing delegate-based matcher classes

The delegate-based matchers implement methods that a delegate interface in the `gw.api.logicalmatch` package defines. The delegate interface depends on the type of the object to match.

- For effective-dated subobjects in a `PolicyPeriod` graph, the matchers use the `EffDatedLogicalMatcher` interface. This interface extends the `LogicalMatcher` interface.
- For all other entity types, the matchers use the `LogicalMatcher` interface.

The base comparison is in the `PCBeanMatcher` class. This class compares the `Id` property and, for effective-dated subobjects in a `PolicyPeriod` graph, the `FixedID` property.

To define a new matcher, extend the relevant abstract class that PolicyCenter provides as a base implementation of each matcher interface in the `gw.api.logicalmatch` package:

- For effective-dated subobjects in a `PolicyPeriod` graph, extend the `AbstractEffDatedPropertiesMatcher` class.
- For all other entity types, extend the `AbstractPropertiesMatcher` class.

In your new matcher class, override the abstract property definition for the `IdentityColumns` property. You use the set of fields that uniquely identify an entity to create the `IdentityColumns` property of the matcher.

To assist the matching task, the `LogicalMatcher` interface provides the `genKey` method to generate a unique key for an object. In the abstract implementations of the `LogicalMatcher` interface, the `genKey` method uses the `IdentityColumns` property to create the unique key. Two objects match if this unique key is the same for both objects. If the keys for two objects do not match, the objects definitely do not refer to the same item.

If the keys for two objects match, the objects might or might not refer to the same item. To confirm whether the two objects are the same, PolicyCenter calls the `isLogicalMatchUntyped(KeyableBean bean)` method, which optionally performs more checks. These additional comparisons could include whether a set of parent foreign keys also match. For example, two `VehicleDriver` entity instances match if the foreign keys to `PersonalVehicle` and `PolicyDriver` also match.

The implementation of the delegate-based matchers in the base configuration relies on properties on the entity, including parent foreign key links, to determine whether two objects match.

General guidelines for writing new matchers are described below.

- You can match on whatever properties you want, but using properties backed by database columns is recommended over using virtual properties.

Avoid matching on non-database-backed properties, such as dynamically generated Gosu properties. These types of properties are marked as `(virtual property)` in the *Data Dictionary*. For example, the `CostNew` property of a `PersonalAuto` instance is a virtual property. Contact Guidewire Customer Support for guidance before implementing matchers that use virtual properties.

- Column values must be immutable values that never change over the life of the object. For example, an automobile VIN does not change over time. The only reason for editing a VIN is to correct a data entry error.

APIs for matcher classes

The public API that supports matcher classes consists of two interfaces and two abstract classes that implement the interfaces. These interfaces and classes are in the `gw.api.logicalmatch` package. The two interfaces extend the `LogicalMatcher` interface, which provides the following methods.

`isLogicalMatchUntyped`

Returns `true` if the argument object is logically the same as the calling object and returns `false` otherwise.

`genKey`

Generates a logical key that identifies the calling object.

The `KeyBasedLogicalMatcher` interface provides an additional `isLogicalMatch` method that supports strongly typed matching of objects. The abstract `AbstractPropertiesMatcher<T extends KeyableBean>` class implements this interface. Extend this class to match objects of a class that is not effective-dated.

The `EffDatedLogicalMatcher` interface provides an additional `findMatchesInPeriodUntyped` method that supports matching effective-dated objects. Effective-dated entity types implement this interface. The abstract `AbstractEffDatedPropertiesMatcher<T extends EffDated>` class extends the `AbstractPropertiesMatcher<T extends KeyableBean>` class and implements this interface. Extend this class to match objects of a class that is effective-dated.

See the Javadoc for descriptions of the classes and methods and the lists of classes that implement the interfaces.

Customizing the classes that perform matching

The `PCBeanMatcher` object always uses the `gw.api.logicalmatch.LogicalMatcher` delegate if one is available. Note that effective-dated objects use the `EffDatedLogicalMatcher` interface, which is a subtype of `LogicalMatcher`. Each effective-dated entity type in the base configuration provides a delegate class that implements `EffDatedLogicalMatcher`. If you need to change the matching behavior for generating a list of differences, you can edit that class.

Copier classes

In many cases, PolicyCenter uses a copier as well as a matcher. For example, jobs other than submission jobs need to do the following process, which is known as `DuplicateAdd`.

1. Generate a policy difference between two branches to find added entity instances.
2. Use matchers to check whether new entity instances have a match elsewhere in the graph.
3. If a match exists, use copiers to ensure that the matches are fully synchronized, as defined by the entity type.

A copier class is able to completely duplicate an entity instance.

IMPORTANT If you write a new matching class, you must write a new copier class. If a copier class does not exist or does not correctly implement the duplication of the entity instance, a task such as a `DuplicateAdd` does not work as expected. PolicyCenter might extend an existing entity, but lose new property values by not copying them.

In a typical policy difference operation, the difference generation is read-only. The difference generation itself does not need to copy data from one entity to a duplicate of that entity.

Each entity type that uses a copier class extends that class from the abstract class `gw.api.copy.Copier` or one of its subclasses. For example, the copier class for the `PersonalAuto` entity type is `PersonalVehicleCopier`.

If you create a new entity type that the policy graph uses, you need to create a copier class for that entity type. Similarly, if you extend an existing entity, you must determine whether that extension affects the copier class. If your extension includes a database-backed property, you typically need to copy that property. For example, if you extend the `PersonalAuto` entity type, consider whether you need to modify the `PersonalVehicleCopier` class.

See also

- *Application Guide*

Bean matcher classes

PolicyCenter provides the bean matcher class `gw.api.diff.PCBeanMatcher`. Most entity types implement one of the matcher delegates and do not have explicit code in `PCBeanMatcher`. The `PCBeanMatcher` class contains code to match only a few entity types, most notably including `Cost` objects.

The policy difference system always initially uses a `PCBeanMatcher` object to perform the base comparison of the `Id` property and, for effective-dated subobjects in a `PolicyPeriod` graph, the `FixedID` property. Similarly, the PolicyCenter out-of-sequence handling system initially calls `PCBeanMatcher` methods.

This class compares two objects and determines if they are equal by performing the following checks:

- Comparing the object references for equality
- Comparing the `fixedID` property
- Using the delegate-based matcher `isLogicalMatchUntyped` method
- For a very few entity types, comparing specific property values

If any one of these checks returns a value of `true`, the two entity instances are the same and the bean matcher performs no further checks.

If two entity instances have the same fixed ID (`fixedID` property), those entity instances represent the same item. If the fixed IDs do not match, PolicyCenter performs additional checks. For example, two cars are the same if their vehicle identification numbers (VINs) match.

These comparisons are typically performed by the delegate-based matcher method that the bean matcher calls. The matchers that PolicyCenter entity types provide handle common cases. However, you can customize the logic in the delegate-based matcher to support your business needs.

Important files for customizing policy differences

The most important files for you to customize policy change differences are listed in the following table.

File name	Example in the base configuration	Products and LOBs that use the file	Description
<code>PolicyPeriodDiffPlugin.gs</code>	n/a	All	Functions to filter the list of <code>DiffItem</code> objects and compare entities.
<code>LOBDiffHelper.gs</code>	<code>PADiffHelper.gs</code>	All	Contains the helper functions to filter and add <code>DiffItem</code> objects for this line of business.
<code>PCBeanMatcher.gs</code>	n/a	All	For multi-version policy comparisons and other contexts for comparing two branch graphs, this file compares properties in entities to determine whether two entities are the same. For example, it checks if cars are the same by checking their VIN number.
<i>file names vary</i>	n/a	All	Delegate-based matcher files for each type.
<code>difftree.xsd</code>	n/a	All	The XSD for node generation configuration XML files.
<code>PRODUCTDiffTree.xml</code>	<code>PADiffTree.xml</code>	All	The node generation configuration XML file for each product. You must have one node generation configuration file for each product, such as PA (personal auto) or WC (Workers' Comp). For products that have multiple lines of business, such as commercial package, they share one XML file.
<code>DiffTreePanelSet.pcf</code>	n/a	All	Displays the differences tree based on settings in the XML file.

File name	Example in the base configuration	Products and LOBs that use the file	Description
DifferencesPanelSet.pcf	n/a	All	Displays the differences tree based on settings in the XML file.

Filtering difference items after creation of database records

One of the two difference item strategies is to compare a branch to its based-on branch. After PolicyCenter generates the full list of differences at the database level, it calls the policy period difference plugin method `filterDiffItems` to remove irrelevant differences.

In your plugin, you can add or change the rules for filtering differences. You might use simple collection Gosu methods to remove items, such as the following to remove all `TerritoryCode` information that was added or removed from one branch to the other.

```
items.removeAll(items.findAll(\ d -> (d.Object typeis TerritoryCode)
    and (d.Add == true or d.Remove == true)))
```

Alternatively, you can define your logic to apply only to certain conditions, such as the reason code that is a parameter to the method. It is an enumeration called `DiffReason` containing values such as `TC_POLICYREVIEW`, `TC_INTEGRATION`, `TC_MULTIVERSIONJOB`, `TC_PREEMPTION`, `TC_COMPAREJOBS`.

For example, the following code filters differences for out of sequence jobs.

```
if (reason == DiffReason.TC_APPLYCHANGES) {
    diffHelper = new DiffHelper(reason, null, null)
    diffItems = diffHelper.filterDiffItems(diffItems)
}
```

You may want to update or add to the logic for each line of business in the helper files such as `PADiffHelper` for personal auto.

After filtering the generated difference list, if PolicyCenter needs to display the differences to users, PolicyCenter passes the list to the difference tree generation system for furthering processing. You configure what differences the tree view displays and how the differences appear by customizing difference tree configuration XML files.

See also

- “Customizing differences for new lines of business” on page 474
- “Customizing personal auto line of business” on page 475

Difference tree XML configuration

The full set of differences might be more than you want users to see in the user interface. PolicyCenter needs to perform the complex actions like merging changes or integration with external systems. Those parts of the system might need to track a large and complete list of difference items. To configure what differences users can view, you use a difference tree configuration XML file.

By default, PolicyCenter defines one difference tree XML configuration file for each product. PolicyCenter manages difference tree configuration XML files by product, not by line of business. Products that have multiple lines of business, such as commercial package, share one XML file. You can customize these files, and you can create additional files if you create new products. Even if a product has no lines of business, you must add a new difference tree XML configuration file.

In practice, a new line of business is one of the following items.

- Part of a new product, in which case you add a new XML file
- Part of an existing product, in which case you customize an existing XML file

A difference tree XML file defines the following situations.

- Which entities to display with changes to properties
- Which entities can have child objects in the tree
- How to group changes in an intuitive way using sections
- How to display the context of a property or entity by showing ancestor groups in the tree structure

This is primarily a user interface customization. PolicyCenter only uses this XML file to display difference items. Although you can use this to hide some difference items in the user interface, hiding these differences do not fundamentally change the data used in other parts of the application. For example, to apply preemptions or out-of-sequence policy changes, PolicyCenter merge differences that you choose not to display.

However, it is generally best to do as much filtering as possible in the XML configuration. Avoid writing additional Gosu code in the difference helper classes (`DiffHelper` subclasses) to filter the list of difference items.

There are three ways to filter difference item display in the XML configuration.

- An `ExcludedProperty` element in XML excludes certain properties from displaying. An example of this is in `IMDiffTree.xml`.
- If you do not define an entity in the XML file, PolicyCenter will not display it.
- On each `<Entity>` element in the XML file, there are attributes `showadds`, `showremoves`, and `showchanges`. Use those to suppress difference adds, removes, and changes.

Do as much of your filtering in the XML files as possible. However, note that the difference tree XML file primarily customizes the user interface.

PolicyCenter takes the already-generated list of difference items and this XML file and constructs a tree that represents nodes in the tree that appears in the user interface.

The following steps describe how PolicyCenter uses various objects to create the difference user interface.

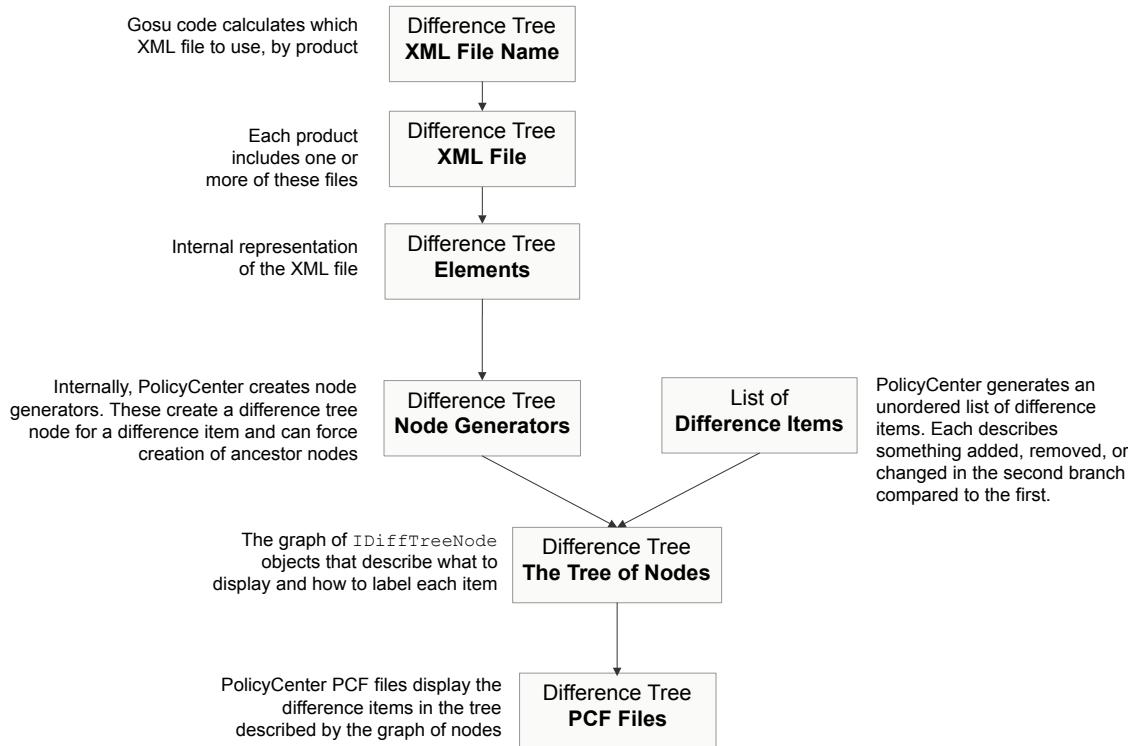
1. Some application flow determines it is time to recalculate the differences user interface.
2. That code calls the policy period difference plugin method `recalculateRootNode`. This method must do whatever is necessary to build the difference tree and return it. More precisely, it returns the root node type, which is `RowTreeRootNode`. Either base your code on the built-in implementation or modify the built-in implementation. The built-in implementation gets the file name of the XML configuration file and then builds the tree. The following steps describe this process in more detail.
3. Gosu code calculates which XML file to use. By default, the built-in implementation just uses the line of business and returns a specific file. To change the default logic, customize the policy period difference plugin method `getDiffTreeConfig`. In the base configuration, this method exists only on the implementation class for the built-in plugin implementation. You can put complex logic into this. For example, you might dynamically change which XML file to use for your product based on one or more of the following conditions.
 - Whether the user is an underwriter or a producer and show different levels of detail to each.
 - The difference reason. Is this a multiple version job or comparing a policy to a history revision?
 - Line of business.
 - Other application context.
4. PolicyCenter loads the XML file based on its file name. It specifies how to present entities, property changes, and how to label sections in the user interface. This file name is an argument to the constructor for the `DiffTree` class, which controls much of this flow. You can view the `DiffTree.gs` file in Studio.
5. PolicyCenter creates an internal in-memory representation of the difference tree XML elements.
6. PolicyCenter creates a tree of node generators. A node generator knows how to generate a node in the final tree. The node generator tree is roughly parallel to the XML element tree. However, there are important

differences. The most important differences between the XML structure and the node generator tree structure are described below.

- In most cases property changes naturally appear under the entities they are on. Because of this, the format implicitly creates node generators underneath entities that generate nodes for changed properties. Because of this, compared to the input XML file structure, the generator tree has additional objects. (There are ways to customize the property behavior, such as suppressing properties or having them display in other places in the tree structure.)
 - For any entity that you identify in the XML file, in the typical case PolicyCenter creates three node generators. One generator knows how to display a row identifying an added object. One generator knows how to display a row identifying a removed object. One generator (the `EntityNodeGenerator`) is primarily a container for descendants in the tree. For example, changed properties for that entity appear underneath this section.
7. PolicyCenter iterates across the list of difference items and finds the appropriate node generator for that difference item using the following information.
- The type of the object that changed
 - Whether the object was added, removed, or changed
 - If a property changed, what is the property name
- If PolicyCenter does not find a node generator for that information, then PolicyCenter skips this difference item. It does not appear in the difference tree user interface.
8. For each difference item, PolicyCenter asks the appropriate node generator to generate a new node in the tree of tree nodes. Each node is an object that implement the `IDiffTreeNode` interface. For example, if a property changed, the node object contains information about how to display that specific property change.
9. Before proceeding to the next difference item, PolicyCenter must determine where in the destination tree to put the new node. For example, suppose the difference item fixes an incorrect drivers license number for a driver on car. Presumably you want this change to appear underneath a tree node with a label that describes the driver. If it is important to understand that the driver displays under a specific car, then PolicyCenter must create a node for that label too, one level above the driver.
- PolicyCenter first checks to see if the parent container for this new node already exists in the actual difference tree yet. If it does exist, PolicyCenter sets the new node to point to its parent container. If it does not exist, PolicyCenter asks the appropriate node generator for the parent type to generate the container node. (It knows what generator to use by looking at the tree of node generators, which have parent links that lead up eventually to the root.) This process recursively ensures that all new nodes are contained within ancestor nodes that tell the user the context of the new change.
- For example, suppose there are three changes on a car for an auto policy. For the first property change, PolicyCenter asks the node generator to create the container node representing the car. For additional property changes on that car, PolicyCenter can reuse the existing parent container node that represents the car. All three property changes appear as child nodes underneath the car.
- The result is a complete tree of `IDiffTreeNode` objects that describe the structure of the difference tree.
10. The user interface PCF files use the tree to display the difference tree using a tree widget.

The following diagram describes the basic flow of the difference tree creation.

Difference Tree Creation



There is not a one-to-one relationship between the difference items and the nodes in the final output tree. The XML file describes what information appears and how to structure it. The number of input difference items is independent of the number of final output nodes.

- It is common for the number of nodes in the final difference tree to exceed the number of difference items. This is because PolicyCenter creates nodes to describe ancestors in the tree to what entities changed. Also, some nodes describe groups solely for display purpose (these are called “sections”).
- The converse could also be true. For example if there is no reference to that entity in the difference tree XML file or you filter out some costs. Thus, there could be more difference items than nodes in the final tree.

For example, consider a single difference item representing a change of driver on a car. The difference tree might include multiple nodes that describe the change as well as multiple levels of context of the change , as shown in the following line. The example shows one difference item but four nodes in the final output tree. The four nodes include nodes that describe the ancestors of the node that describes the change. Users can see each change in its full context.

Root of the line of business → A label that contains “Vehicles” → A specific car → The change of driver

Editing the difference tree XML files

PolicyCenter includes difference tree XML configuration files for the built-in products. This is the primary mechanism for customizing how to show differences to users. Edit these files to modify the default behavior. Create additional XML files as desired.

To access the difference tree XML files in Studio, in the **Project** window, navigate to **configuration→config→resources→diff**. To view the XSD for the difference tree XML files, in the **Project** window, navigate to **configuration→config→resources→diff→schema**. Open the `difftree.xsd` tree.

The following is an example of a personal auto XML difference tree configuration file.

```

<DiffTree xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="schema/difftree.xsd">
  <Section label="displaykey.Web.Differences.LOB.Common.PolicyInfo">
    <RootProperties includefromtype="PolicyPeriod" sortorder="1"/>
    <RootProperties includefromtype="EffectiveDatedFields" sortorder="2">
      <PropertyDisplay propertyname="OfferingCode" value="ENTITY.getOfferingName(VALUE)"/>
    </RootProperties>
    <RootEntity type="PolicyPriNamedInsured">
      label="displaykey.Web.Differences.LOB.Common.PolicyPriNamedInsured(ENTITY.DisplayName)"
      sortorder="3"/>
    <RootEntity type="PolicySecNamedInsured">
      label="displaykey.Web.Differences.LOB.Common.PolicySecNamedInsured(ENTITY.DisplayName)"
      sortorder="4"/>
    <RootEntity type="PolicyAddlInterest" showadds="false" showremoves="false">
      label="displaykey.Web.Differences.LOB.Common.PolicyAddlInterest(ENTITY.DisplayName)"
      sortorder="5"/>
    <RootEntity type="PolicyAddress">
      label="displaykey.Web.Differences.LOB.Common.PolicyAddress(ENTITY.AddressType)"
      sortorder="6"/>
    </Section>
    <Section label="displaykey.Web.Differences.LOB.PA.Drivers">
      <RootEntity type="PolicyDriver">
        label="displaykey.Web.Differences.LOB.Common.PolicyDriver(ENTITY.DisplayName)"/>
    </Section>
    <Section label="displaykey.Web.Differences.LOB.PA.Vehicles">
      <RootEntity type="PersonalVehicle">
        <SubSection label="displaykey.Web.Differences.LOB.Common.Coverages" sortorder="1">
          <Entity type="PersonalVehicleCov" parentpath="ENTITY.PersonalVehicle"/>
        </SubSection>
        <SubSection label="displaykey.Web.Differences.LOB.Common.Modifiers" sortorder="2">
          <Entity type="PAVehicleModifier" parentpath="ENTITY.PAVehicle">
            <Properties includefromtype="PAVehicleModifier" parentpath="ENTITY">
              <PropertyDisplay propertyname="TypeKeyModifier" value=""
                if (ENTITY.Pattern == "PAPassiveRestraint") {
                  return (VALUE as PassiveRestraintType).DisplayName
                } else if (ENTITY.Pattern == "PAAntiTheft") {
                  return (VALUE as AntiTheftType).DisplayName
                } else {
                  return VALUE as String
                }
              </PropertyDisplay>
            </Properties>
          </Entity>
        </SubSection>
        <Entity type="VehicleDriver" parentpath="ENTITY.Vehicle">
          label="displaykey.Web.Differences.LOB.PA.AssignedDriver(ENTITY.DisplayName)" sortorder="3"/>
        <Entity type="PAVhicleAddlInterest" parentpath="ENTITY.PAVehicle">
          label="displaykey.Web.Differences.LOB.Common.PolicyAddlInterestDetail(ENTITY.DisplayName,
          ENTITY.AdditionalInterestType)" sortorder="4">
            <Properties includefromtype="PAVhicleAddlInterest" parentpath="ENTITY">
              <PropertyDisplay propertyname="AdditionalInterestType" />
            </Properties>
          </Entity>
        </RootEntity>
      </Section>
      <Section label="displaykey.Web.Differences.LOB.Common.LineCoverages">
        <RootEntity type="PersonalAutoCov"/>
      </Section>
    </DiffTree>
  
```

Notice that this relatively small file describes all the possible changes to the personal auto line of business. One reason it is so small is that it does not have to specify all the properties on each entity. For property changes, the default node generation code implicitly creates nodes in the difference tree underneath each entity. You can customize the behavior with elements and attributes to omit properties.

Difference tree element

The root element in the XML is a `<DiffTree>` element. It represents the root of the difference tree. The root element contains one or more labeled sections, each represented by the `<Section>` XML element.

Section element

A section element (`<Section>`) is essentially a container. A section groups together various other properties and entities. For example, the personal auto XML has several sections.

- A section for policy information
- A section for drivers
- A section for vehicles

The section contains a label expression to determine what to display as the label for the section. The label must be a valid Gosu expression that returns a `String`.

Sections correspond to top level label nodes in the resulting difference tree such as **Policy Info** or **Locations and Buildings**. Sections can have child elements of three types.

- `RootEntity`
- `RootProperties`
- `Section`

```
<Section label="displaykey.Web.Differences.LOB.PA.Drivers">
  <RootEntity type="PolicyDriver"
    label="displaykey.Web.Differences.LOB.Common.PolicyDriver(ENTITY.DisplayName)"/>
</Section>
```

Root entity element

The root entity element (`<RootEntity>`) corresponds to a particular type of Guidewire business entity, such as **PolicyLocation** or **Building**. Root entity elements require a type attribute specifying the type for which this element applies. These elements correspond to all nodes in the generated tree of that particular type. For example, the node for a particular added **PolicyLocation** or the container **Building** node for a property change on a **Building**.

The root entity element has the following Boolean attributes.

- `showadds` – Specifies whether to display added Guidewire entities of this type. The default is `true`. Set it to `false` to suppress the default behavior.
- `showremoves` – Specifies whether to display removed Guidewire entities of this type. The default is `true`. Set it to `false` to suppress the default behavior.
- `showchanges` – Specifies whether to use this entity as a container for property change difference items. The default is `true`. Set it to `false` to suppress the default behavior. However, setting the `showchanges` attribute to `false` on a root entity element does not affect any explicitly defined child `<Properties>` elements in other parts of the XML hierarchy. Also, setting the `showchanges` attribute to `false` on a root entity element does not affect any explicitly defined child `<Properties>` elements directly under a `<RootEntity>` element.

The root entity element also has a `label` attribute you can optionally use to customize what label to display for entities of this type. The expression can access a symbol `ENTITY`, which corresponds to the entity for the node. This symbol automatically has the appropriate Gosu type for the entity. If you do not set this attribute (or it is `null`), the difference tree node uses the entity's `DisplayName` property.

These elements can have children of three types.

- `EntityElement`
- `PropertiesElement`
- `SubSectionElement`

```
<RootEntity type="PolicyDriver"
  label="displaykey.Web.Differences.LOB.Common.PolicyDriver(ENTITY.DisplayName)"/>
```

Entity element

An entity element (`<Entity>`) is exactly the same as a `RootEntity` element except that these elements have an ancestor element of type `RootEntityElement` or `EntityElement`. Since the ancestor entity element itself requires an entity at node generation time, these elements must additionally supply a `parentpath` expression.

This `parentpath` expression is a Gosu expression that returns an entity of the same type as the ancestor entity element. The expression can access a symbol `ENTITY`, which corresponds to the entity for the node. This symbol automatically has the appropriate Gosu type for the entity.

```
<Entity type="PersonalVehicleCov" parentpath="ENTITY.PersonalVehicle"/>
```

Subsection element

Separate from the section element, the XML format supports the subsection element (`<SubSection>`). It is the same as `<Section>` except that it must have an ancestor element of type `RootEntityElement` or `EntityElement`. Because a subsection is underneath an entity in the hierarchy, it can have different types of children.

Subsection elements display differently than sections. Subsections do not have the same gray line as sections, and subsections appear mostly like other tree elements, but with an additional level of subsections. Use subsection elements to organize difference data into labeled folder-like sections. For example, you could organize the properties on an entity under a label named **Properties** by putting a `<Properties>` element underneath the `<SubSection>` element.

Subsection elements can have children of three types.

- `EntityElement`
- `PropertiesElement`
- `SubSectionElement`

```
<SubSection label="displaykey.Web.Differences.LOB.Common.Coverages" sortorder="1">
  <Entity type="PersonalVehicleCov" parentpath="ENTITY.PersonalVehicle"/>
</SubSection>
```

Root properties element

The root properties element `<RootProperties>` corresponds to the properties or fields on a particular type of entity. These elements require an `includefromtype` attribute that specifies the type for which this element applies. This type correspond to property change nodes in the generated tree on the specified type.

The root properties elements can contain the following elements.

- Property display elements (`<PropertyDisplay>`), which override display logic for specific properties on the specified type.
- Excluded property elements (`<ExcludedPropertyElements>`), which define properties that you want to exclude from node generation.

```
<RootProperties includefromtype="EffectiveDatedFields" sortorder="2">
  <PropertyDisplay propertyname="OfferingCode" value="ENTITY.getOfferingName(VALUE)"/>
</RootProperties>
```

Properties element

A properties element (`<Properties>`) is exactly the same as a `RootPropertiesElement` except that these elements have an ancestor element of type `RootEntityElement` or `EntityElement`.

Because the ancestor entity element itself requires an entity at node generation time, these elements must additionally supply a `parentpath` expression. This `parentpath` expression must be a Gosu expression that returns an entity of the same type as the ancestor entity element. The expression can access a symbol `ENTITY`, which corresponds to the entity for the node. This symbol automatically has the appropriate Gosu type for the entity.

```
<Properties includefromtype="PAVehicleModifier" parentpath="ENTITY">
  <PropertyDisplay propertyname="TypeKeyModifier" value=
    if (ENTITY.Pattern == "PAPassiveRestraint") {
      return (VALUE as PassiveRestraintType).DisplayName
    } else if (ENTITY.Pattern == "PAAntiTheft") {
      return (VALUE as AntiTheftType).DisplayName
    } else {
      return VALUE as String
    }
  </PropertyDisplay>
</Properties>
```

```
"/>
</Properties>
```

Property display element

The property display element (`<PropertyDisplay>`) describes the display information for a specific property.

This element has the following attributes.

- `propertynname` – Specifies which property name to display.
- `sortorder` – An optional attribute to define sort order of property changes. Lower values appear visually first. The value 1 is the highest priority sort order.
- `label` – An optional Gosu expression that specifies the label of the node. The label expression takes two arguments, the entity and the property, and returns the `String` to use for the `Label` column when generating nodes for this property. If you do not specify this attribute or set it to `null`, the default behavior is described below. The first rule in the list that applies is used.
 - Use a matching `CovTermPattern` name if applicable.
 - Use a display key of the form `displaykey.entity.TYPE_NAME.PROPERTY_NAME` if one exists.
 - Use the property's `Name` property as defined in the data model. However, PolicyCenter inserts space characters before each capitalized letter other than the initial character.
- `value` – An optional Gosu expression that determines what text to display. The expression takes three arguments, an entity, a property name, and a value. It returns the `String` to use for the property when generating nodes. PolicyCenter calls this expression once for each branch that it compares. If comparing two branches, PolicyCenter calls this expression once to display `Value1` (typically the left column) and once to display `Value2` (typically the right column). If you do not specify this attribute or set it to `null`, the default behavior is described below. The first rule in the list that applies is used.
 - Use a matching `CovTermPattern.DisplayValue` display value property, if applicable.
 - Display Boolean values as either `Yes` or `No`.
 - Typekeys use their `Name` property instead of the `Code` or `Description` properties. You can configure this value with a display key.
 - Dates use the format in the system configuration parameter `DefaultDiffDateFormat`. The default format is `"short."`
 - Convert the value to a `String`.

If you provide a value Gosu expression, none of the default display logic applies for Boolean values, typekeys, or dates. If you want this display logic, you must explicitly replicate this display logic in the Gosu expression.

```
<Properties includefromtype="PAVehicleModifier" parentpath="ENTITY">
<PropertyDisplay propertynname="TypeKeyModifier" value=""
  if (ENTITY.Pattern == "PAPassiveRestraint") {
    return (VALUE as PassiveRestraintType).DisplayName
  } else if (ENTITY.Pattern == "PAAntiTheft") {
    return (VALUE as AntiTheftType).DisplayName
  } else {
    return VALUE as String
  }
"/>
</Properties>
```

Excluded property element

The excluded property element (`<ExcludedProperty>`) excludes a specified property from node generation.

Specify the property to exclude in the `propertynname` attribute.

The following example shows an excluded property inside a properties element.

```
<Properties includefromtype="IMAccountsReceivable" parentpath="ENTITY" sortorder="2">
  <ExcludedProperty propertynname="AccountsRecNumber"/>
</Properties>
```

Customizing differences for new lines of business

If you create a new line of business, you must perform the following tasks to support the difference engine.

- Using some appropriate abbreviation prefix for the line of business (*LOB*), create a new Gosu class called *LOBDiffHelper*. In this file, add all of the methods that help you add and filter difference items. Specifically, create a method in this class called *addDiffItems* and *filterDiffItems*.
- In your line of business, create a new method *createPolicyLineDiffHelper* that creates and returns an instance of your new difference helper subclass.
- Create a new difference tree XML file that describes the structure of any new products. Remember that PolicyCenter configures the XML files one for each product, not one for each line of business. Even if a product had no lines of business, you would need a new XML file. In practice, a new line of business is one of the following items.
 - Part of a new product, in which case you add a new XML file
 - Part of an existing product, in which case you customize an existing XML file
- Because there are several reasons for calculating differences, you can vary your application logic based on the difference reason. The difference reason is a *DiffReason* enumeration passed to the *PolicyPeriodDiffPlugin* plugin method *compareBranches*. Choices include *TC_INTEGRATION*, *TC_MULTIVERSIONJOB*, *TC_PREEMPTION*, *TC_POLICYREVIEW*, *TC_COMPAREJOBS*.
- You might want to also customize the *PCBeanMatcher* class, depending on what changes you made.

In your difference helper class method *addDiffItems*, create and add the *DiffItem* objects that you want to display in the multi-version difference screen. There are utility methods in the *DiffUtils* class for comparing, generating, and filtering difference items. Follow the pattern of the built-in lines of business and products for guidance.

You may have to manually add entities not automatically recognized from the compare functions in *DiffUtils*, for instance *Effective* and *Expiration* date differences between versions.

You can create *DiffItems* manually for these objects by creating an instance of one of the following classes that are subclasses of *DiffItem*. For instance, *DiffAdd*, *DiffRemove*, and *DiffSlice*.

Filtering difference items

To support filtering difference items from database-generated difference items, such as those used by the policy change job, perform the following tasks.

1. In your new Gosu class *LOBDiffHelper*, create a method that overrides *filterDiffItems*.
2. In *filterDiffItems*, remove all difference items you do not want to display. Depending on what you are doing, you might be able to use simple logic using blocks.
3. Call the *filterDiffItems* method from *filterDiffItems* in *PolicyPeriodDiffPlugin*, as shown in the bold line in the example below.

```
// Filter diffs by LOB if this is not for integration or out of sequence jobs
if ( reason != DiffReason.TC_INTEGRATION
    and reason != DiffReason.TC_APPLYCHANGES) {
    // Add diffs for PolicyPeriod attributes
    if (currentPeriod.Renewal == null and reason != null) {
        diffItems = addPolicyPeriodDiffItems(currentPeriod, currentPeriod.BasedOn, diffItems)
    }

    // Filter diffs by LOB
    for (line1 in currentPeriod.BasedOn.Lines){
        var line2 = currentPeriod.Lines.firstWhere( \ p -> p.Subtype.Code == line1.Subtype.Code )
        if (line2 != null) {
            diffHelper = line1.createPolicyLineDiffHelper(reason, line2)
            if (diffHelper != null) {
                diffItems = diffHelper.filterDiffItems(diffItems)
            }
        }
    }

    // Remove PolicyPeriod attribute diffs if this is a rewrite job
    if (currentPeriod.Rewrite != null) {
        diffItems.removeWhere( \ d -> d.Bean typeis PolicyPeriod )
```

```
}
```

Because there are several reasons for calculating differences, you can vary your application logic based on the difference reason. The difference reason is a `DiffReason` enumeration passed to the `PolicyPeriodDiffPlugin` plugin method `compareBranches`. Choices are defined in the `DiffReason` typelist and include `TC_INTEGRATION`, `TC_MULTIVERSIONJOB`, `TC_PREEMPTION`, `TC_POLICYREVIEW`, `TC_COMPAREJOBS`.

Customizing personal auto line of business

The existing configuration includes several classes to add and filter differences for personal auto line of business.

In personal auto multi-version differences, PolicyCenter first compares the vehicles, coverages, and cost entities. Next, PolicyCenter generates a list of `DiffItem` objects from the differences.

The Gosu class `PADiffHelper` contains a method `addDiffItems` which calls the `DiffUtil` comparison methods to compare vehicles, coverages, and costs. The `DiffUtil` comparison methods compare and return a list of `DiffItem` objects from differences between the entities.

The most important feature of the `DiffUtil` class's `compareFields` method for personal auto multi-version differences because it allows you to compare an array off of a entity. For instance, PolicyCenter compares the vehicles array from the two personal auto line objects that represent two versions in multiple version quote.

Calling the `compareField` with a depth argument set to 2 traverses the vehicle graph to a depth of 2. In other words, for every vehicle compared, it automatically compares properties from that vehicle (depth level 1) and its referenced coverages (depth level 2).

The code that accesses these differences looks like the following statement.

```
paDiffs.addAll(diffUtils.compareField(line1, line2,
    PersonalAutoLine.TypeInfo.getProperty( "Vehicles"), 2))
```

In the `PADiffHelper` class, the `filterDiffItems` method filters out all unnecessarily added difference items. This method is called from the `addDiffItems` method. If you want to customize which properties specific to personal auto policies to be filtered, modify this method.

Also in the `PADiffHelper` class is the `compareBranches` method. PolicyCenter passes this method the two `PolicyPeriod` objects of the two personal auto quotes, the difference reason (`DiffReason`), an enumeration to specify the business context for the difference detection.

The `addDiffItems` method is called from the `compareBranches` method.

```
// Add diffs by line of business
for (line1 in p1.Lines){
    if (line1.Subtype.Code == "PersonalAutoLine") {
        diffs = new PADiffHelper().AutoDiffItems(line1 as PersonalAutoLine,
            line2 as PersonalAutoLine)
        as ArrayList<DiffItem>
    }
}
```

Customize this code if you need different application logic. The `addDiffItems` method in `PADiffHelper` adds individual compared properties. For example, it adds line coverages and vehicle differences using code like that shown below.

```
// Add line coverage diffs
var paDiffs = diffUtils.compareField( line1, line2,
    PersonalAutoLine.TypeInfo.getProperty( "PALineCoverages"), 2)

// Add vehicle diffs
paDiffs.addAll(diffUtils.compareField( line1, line2,
    PersonalAutoLine.TypeInfo.getProperty( "Vehicles"), 2))
```

Refer to the `PADiffHelper.gs` class for full details of what it does for that line of business.

The difference tree for personal auto is contained in the `PADiffTree.xml` file. The user interface for the difference tree is controlled by the XML file.

Methods for calculating differences

In many cases, you can use the base configuration logic to trigger difference calculations for the two basic types of differences. If you need to change the application behavior, you can vary application logic based on the difference reason. You trigger difference calculations by using either of the following two methods on the `PolicyPeriod` entity instance: `getDiffItems` and `compareTo`. Each method takes a `DiffReason` typecode as an argument. If the typecodes that the base configuration provides do not meet your needs, you can extend the `DiffReason` typelist with additional difference reasons.

Differences between a branch and its based-on branch

You can get a list of differences between a branch and its based-on branch, by using `DiffItem` filtering from the policy difference plugin. This filtering is similar to what is done for policy change. This list of differences is generated from the database. Call the `getDiffItems` method on a `PolicyPeriod` entity.

```
myDifferences = myPolicyPeriod.getDiffItems(DiffReason.TC_INTEGRATION)
```

You use code similar to this line in cases such as if you write integration messaging code to notify an external system of changes from Policy Change jobs. You might also use this approach to compare a renewal with the period on which it was based. However, you might prefer to use the `compareTo` method for renewals in some edge cases.

Differences between any two branches

You can get a list of differences between any two branches in the same period, similar to what is done in the multi-version user interface. Call the `compareTo` method on a `PolicyPeriod` entity and pass it another `PolicyPeriod` entity.

```
myDifferences = myPolicyPeriod.compareTo(DiffReason.TC_MULTIVERSIONJOB, anotherPolicyPeriod)
```

You use code similar to this line in cases such as if you write new PCF code to compare arbitrary revisions, or a web service that compares two policy periods.

Part 6

Claim and policy integrations

Claim and policy integration

This topic describes web services, plugin interfaces, and tools for communicating with claim systems such as Guidewire ClaimCenter.

Claim search from PolicyCenter

PolicyCenter calls the `IClaimSearchPlugin` plugin to search for claims against a policy. You implement this plugin interface to use PolicyCenter with an existing claim management system.

PolicyCenter includes a claim search plugin implementation that communicates with Guidewire ClaimCenter and a demonstration implementation that does not call out to anything. You can provide another implementation of this plugin interface to use PolicyCenter with a different claim management system.

This plugin provides the method `searchForClaims`. The method accepts an argument of type `ClaimSearchCriteria`, which contains all the claim properties for which a PolicyCenter user is searching. The argument includes properties such as those listed below.

- `Account` - the account
- `Contact` - the contact
- `DateCriteria` - the date criteria
- `Policy` - the policy entity (check for important properties)
- `PolicyNumber` - the policy number
- `PolicyNumbers` - the array of policy numbers

For more details on these objects, see the *GosuDoc*, the *Java API Reference*, and the *Data Dictionary*.

Your plugin must look up the claims for the dates in the `DateCriteria` object.

This method returns a `ClaimSet` object that encapsulates a list of zero, one, or more claims. Create your array of claims and store the array in the claim set property `Claims`. You can see the *Data Dictionary* to determine the required properties for claims.

There is a built-in implementation class, `GWClaimSearchPlugin`, that calls out to ClaimCenter. The following `searchForClaims` method of that implementation shows how to structure your code for claim search.

```
override function searchForClaims(claimSearchCriteria : IClaimSearchCriteria) : ClaimSet {  
    var claimResult = getClaimsFromExternalSystem(claimSearchCriteria.SearchSpecs)  
  
    if (claimResult == null or claimResult.size() == 0) {  
        throw new NoResultsClaimSearchException()  
    }  
    var result : ClaimSet  
    gw.transaction.Transaction.runWithNewBundle(\ bundle -> {  
        result = new ClaimSet(bundle)  
    })  
    return result  
}
```

```

        for (pcClaim in claimResult) {
            var claim = addClaimToClaimSet(pcClaim, result)
            mapClaimToPeriod(claim, pcClaim.PolicyNumber)
        }
        var claimFilter = new ClaimPolicyPeriodFilterSet(result.Claims)
        result.setClaimsFilter(claimFilter)
    })
    return result
}

```

Your implementation of this plugin must call out to your claim system and generate a new claim set and a series of new claims. Populate all the required properties in the data model for the `Claim` and `Exposure` entities. Refer to the *Data Dictionary* for details.

Get claim details

If PolicyCenter needs more information about a claim, PolicyCenter calls the `getClaimDetailByClaimNumber` method of your claim search plugin to retrieve the claim. This method takes an entire `Claim` entity as a parameter, but the external system might require only the claim number property from that entity. The method returns a `ClaimDetail` object, which contains a small amount of details about the claim in its properties. Refer to the *Java API Reference* or the *Data Dictionary* for the complete list of properties.

Permission to view a claim

There is another method you must implement, called `giveUserViewPermissionsOnClaim`. You are not required to do anything within the method. You can use this if you need to give a PolicyCenter user (specified by name) permission to view the claim (specified by claim number) in the external system.

If the username or claim do not exist in the external claim system and you detect a problem, throw an appropriate `RemoteException` exception.

Policy system notifications

PolicyCenter provides a general architecture for notifications from claim systems to PolicyCenter. The only notification type in the base configuration is to detect large losses.

Receiving a notification of a large loss

A claim system can notify PolicyCenter if a claim reaches a critical threshold, defined by policy type. PolicyCenter 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 PolicyCenter and sends the claim's policy number, loss date, and the total gross incurred. This feature is called "large loss notification."

The PolicyCenter part of this support is a web service called `ClaimToPolicySystemNotificationAPI`. This new web service receives notifications from the ClaimCenter special message transport. If properly configured in ClaimCenter, ClaimCenter calls this web service over the network. If you use a claim system other than ClaimCenter, your claim system can call this web service.

The definitions of the thresholds for what counts as a large loss are defined from within the claim system, not in PolicyCenter.

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.

Claim to policy system notification web service API

The `ClaimToPolicySystemNotificationAPI` web service API has only a single method called `claimExceedsThreshold`.

The method takes the following arguments.

- Loss date object of type `java.util.Date`
- Policy number of type `String`
- Gross total of money of type `MonetaryAmount`
- A transaction identifier of type `String`

The method has no return value.

Policy search web service for claim system integration

A release of ClaimCenter that is compatible with your PolicyCenter release uses search criteria and the `CCPolicySearchIntegration` web service to retrieve policy summaries from PolicyCenter. Typically you do not need to call this web service directly. The base configuration of ClaimCenter calls this web service if you configure ClaimCenter to connect to PolicyCenter for policy search.

If you use a claim system other than ClaimCenter, your claim system can call this web service. However, this web service is written to support a data model very similar to the ClaimCenter model. If you use a claim system other than ClaimCenter, consider writing your own custom web services that directly address integration points between PolicyCenter and your specific claim system.

If ClaimCenter or another claim system needs to identify a policy by a policy number and an effective date, it can call the `searchForPolicies` method in this interface.

The `searchForPolicies` method takes a `CCPCSearchCriteria` object, which contains one or more search criteria such as the producer code (`ProducerCode`), the effective date (`AsOfDate`), or other policy attributes. The method returns an array of `CCPolicySummary` entities.

Be aware that the effective date parameter uses the time component of the date parameter. Either strip the time component to represent midnight (the beginning of the day) or set it to a specific time on that date.

For example, the following Java code calls out to PolicyCenter to search for policies.

```
// Create a new criteria object
CCPCSearchCriteria criteria = new CCPCSearchCriteria();
CCPolicySummary results[]

criteria.setPolicyNumber("54-456353");
Calendar cal = Calendar.getInstance();
cal.set( 2019, 8, 21 );
criteria.AsOfDate = calendar.Time;

results = myCCPolicySearchIntegrationAPI.searchForPolicies( criteria );
```

This array might contain zero items if there are no matches. It could also return more than one match if the request is ambiguous.

Given the returned list of policy summaries, the claim system user might select one of these to get and associate with a claim.

Retrieving a policy

After the claim system knows what policy it needs, the claim system calls the `retrievePolicy` method of the policy search web service to retrieve the policy.

The claim system identifies the requested policy by using the following items.

- Policy number of type `String` which corresponds to the `CCPolicySummary` field `_policyNumber`
- Effective date of type `java.util.Calendar` which corresponds to the `CCPolicySummary` field `_effDate`

An example invocation is shown in the following lines.

```
Calendar asOfDate = Calendar.getInstance();
myEnv = myCCPolicySearchIntegrationAPI.retrievePolicy("ABC:1234", asOfDate);
myCCPolicy = myEnv.CCPolicy;
```

The method returns an instance of `gw.webservice.pc.pcVERSION.ccintegration.entities.Envelope`. The `Envelope` type is an XSD type that encapsulates XML objects that mirror the PolicyCenter policy, but are simplified for use in a claim system. In the built-in implementation, the policy structure is similar to what is needed for ClaimCenter.

The returned policy entity of type `CCPolicy` is defined in the PolicyCenter data model, not the claim system data model. If you need custom properties defined in the external system, add data model extension properties to the PolicyCenter data model version of the `CCPolicy` entity.

Service tiers enable an insurer to provide special handling or value-added services for certain customers, typically high-value customers. For an integration with ClaimCenter, the policy information that ClaimCenter retrieves from PolicyCenter includes the service tier. The integration maps the optional `PolicyCenter.Account.ServiceTier` value to a `ClaimCenter.Policy.CustomerServiceTier`. In PolicyCenter, the typekey values for `ServiceTier` are configurable.

See also

- *Application Guide*

Extend the search criteria

To extend the search criteria, extend the Gosu class `CCPCSearchCriteria`. See the Gosu class `CCPolicySearchIntegration` for how this object is used.

Typecode maximum length and trimmed typecodes

If the length of a product model typecode in PolicyCenter exceeds 50 characters, PolicyCenter trims the code before sending the code to the claim system in a policy search. To perform this trimming, PolicyCenter uses the static method `getCCTypeCodeForPMTyp`e in the `CCProductModelCodeMappingUtil` class. PolicyCenter trims the public ID of the item to create the typecode for both generating the typelist and later mapping values during policy retrieval.

If possible, avoid having codes that exceed the maximum length because you cannot guarantee that a trimmed code is unique. If you must use longer codes, customize the logic of `getCCTypeCodeForPMTyp`e to use a different approach, or add special cases to generate unique values.

Policy search SOAP API

PolicyCenter provides another WSI compliant web service called `PolicySearchAPI`, which contains two methods, `findPolicyPeriodPublicIdByPolicyNumberAndDate` and `findPolicyPublicIdByPolicyNumber`.

The `findPolicyPeriodPublicIdByPolicyNumberAndDate` method uses a policy number and a date to find and return a policy period's public ID.

```
policySearchAPI.findPolicyPeriodPublicIdByPolicyNumberAndDate("123435", DateUtil.currentDate())
```

The method performs the search only against policy periods stored in PolicyCenter.

The `findPolicyPublicIdByPolicyNumber` method uses a policy number to find and return a policy's public ID.

```
policySearchAPI.findPolicyPublicIdByPolicyNumber("123435")
```

The method performs the search only against policies stored in PolicyCenter.

For more general search for policies, use the web service `CCPolicySearchIntegration`.

PolicyCenter exit points to ClaimCenter and BillingCenter

There are screens in the PolicyCenter 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. 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.

In the base configuration, PolicyCenter provides `ExitPoint` widgets that can perform the following operations.

- Open ClaimCenter to directly view and edit claim information. The file `ClaimDetailsCV.pcf` provides a **View in ClaimCenter** button that uses the `ViewClaim` exit point.

To set the URL for the exit point to ClaimCenter, edit the configuration parameter `ClaimSystemURL` in `config.xml`. Set this parameter to the base URL for the server as shown in the sample URL below.

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

- Open BillingCenter to directly view and edit account information. The file `Account_BillingScreen.pcf` provides a **View in BillingCenter** button that uses the `BCAccount` exit point.

To set the URL for the exit point to BillingCenter, edit the configuration parameter `BillingSystemURL` in `config.xml`. Set this parameter to the base URL for the server as shown in the sample URL below.

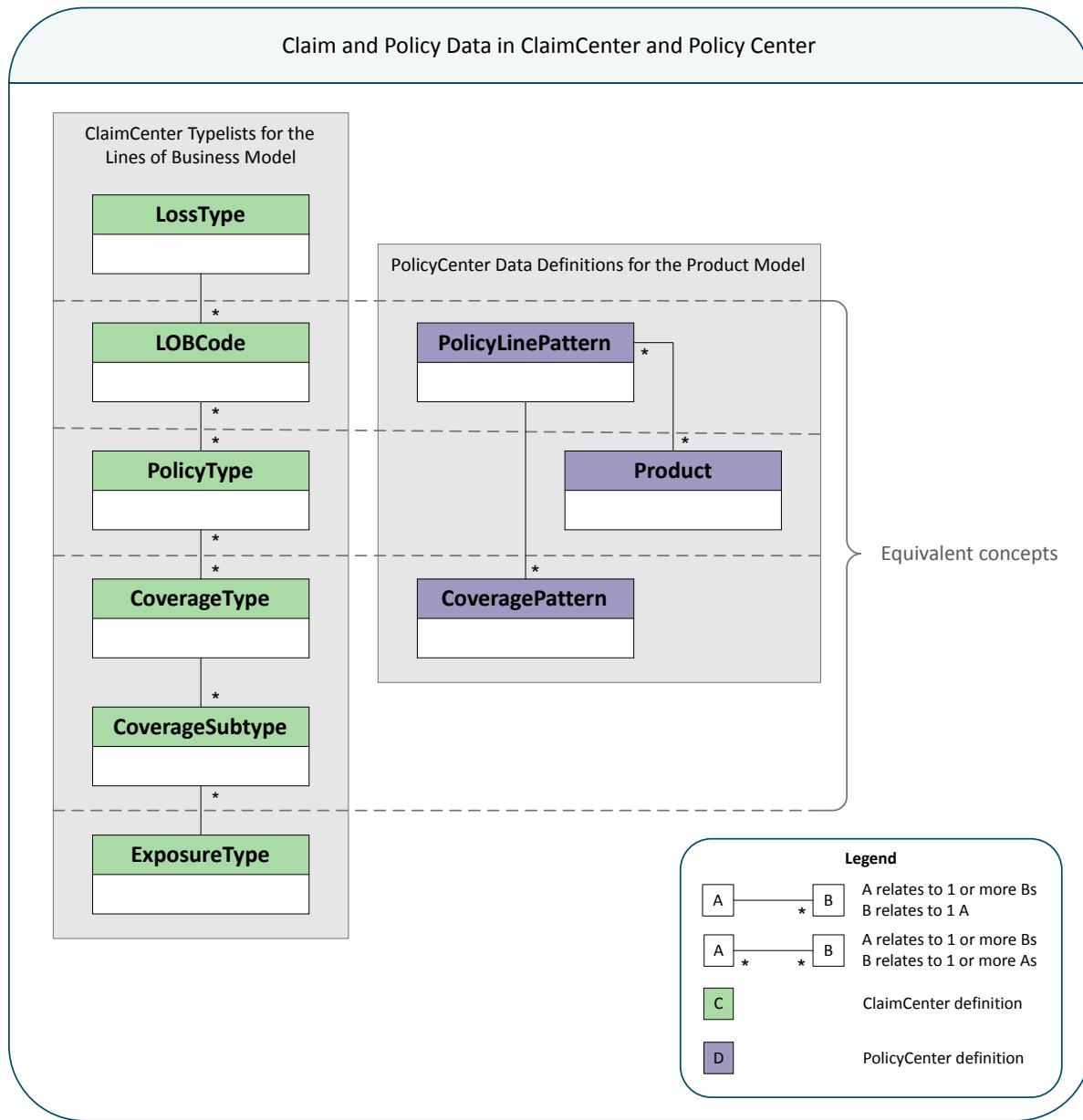
```
http://localhost:8580/bc
```

You can integrate with a system other than a Guidewire core application, such as a billing system that is not Guidewire BillingCenter. If you want to send the system additional parameters, you can add them to the URL configuration parameter for that system. In the case of a billing system, you set the configuration parameter `BillingSystemURL` to the actual URL of the billing system and add any required additional HTTP parameters. If the configuration parameter is absent or is 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 `TYPENAME.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*

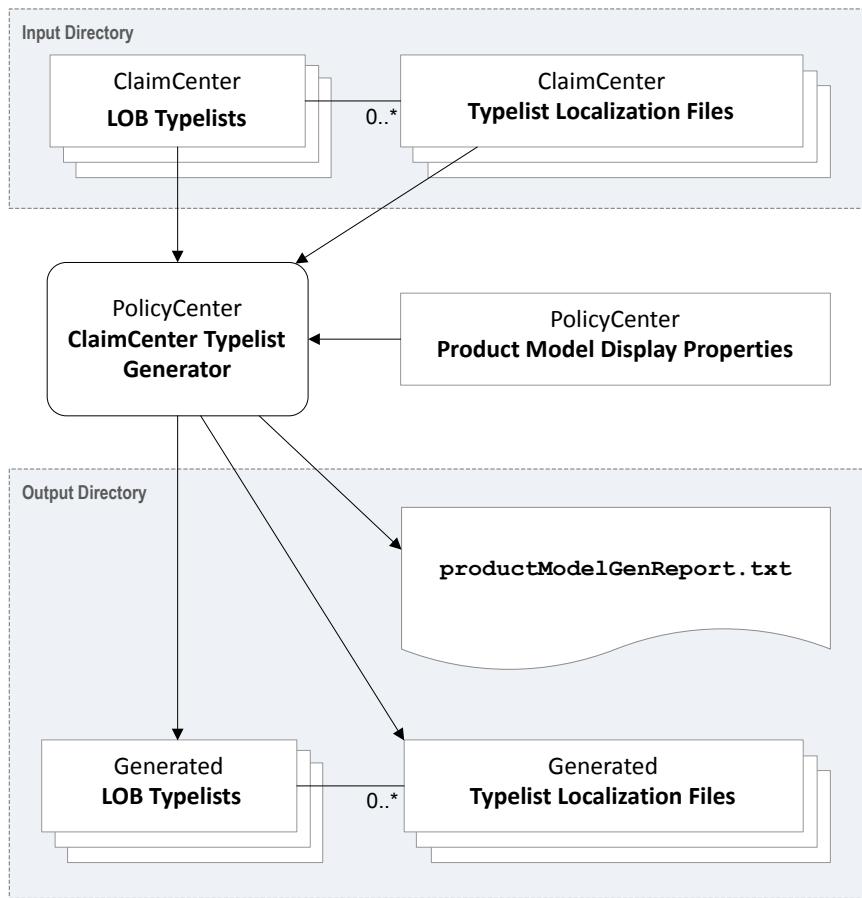
Configuring the ClaimCenter Typelist Generator

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 Studio, open `ProductModelTypelistGenerator.gs` in the `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

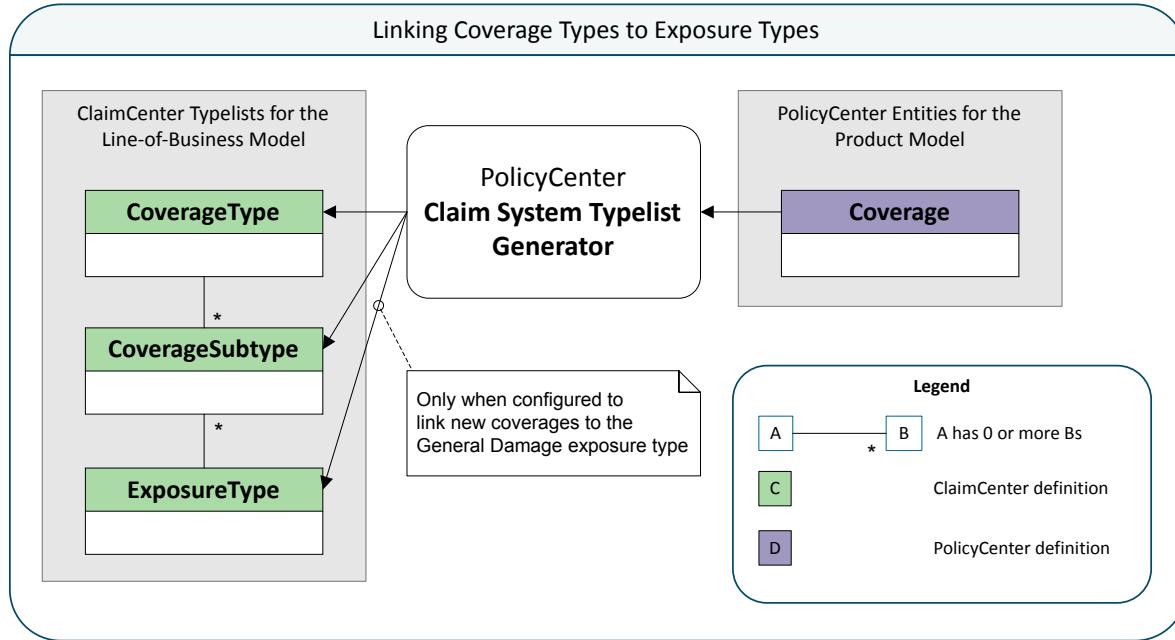
The ClaimCenter Typelist Generator can use the same directory for the input and output directories. The generator reads input files when it starts and writes output files when it finishes. When the input and output directories are the same, your typelist and properties files are overwritten with generated 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 489 and generate the typelists
2. “Copy generated files to ClaimCenter” on page 492

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.

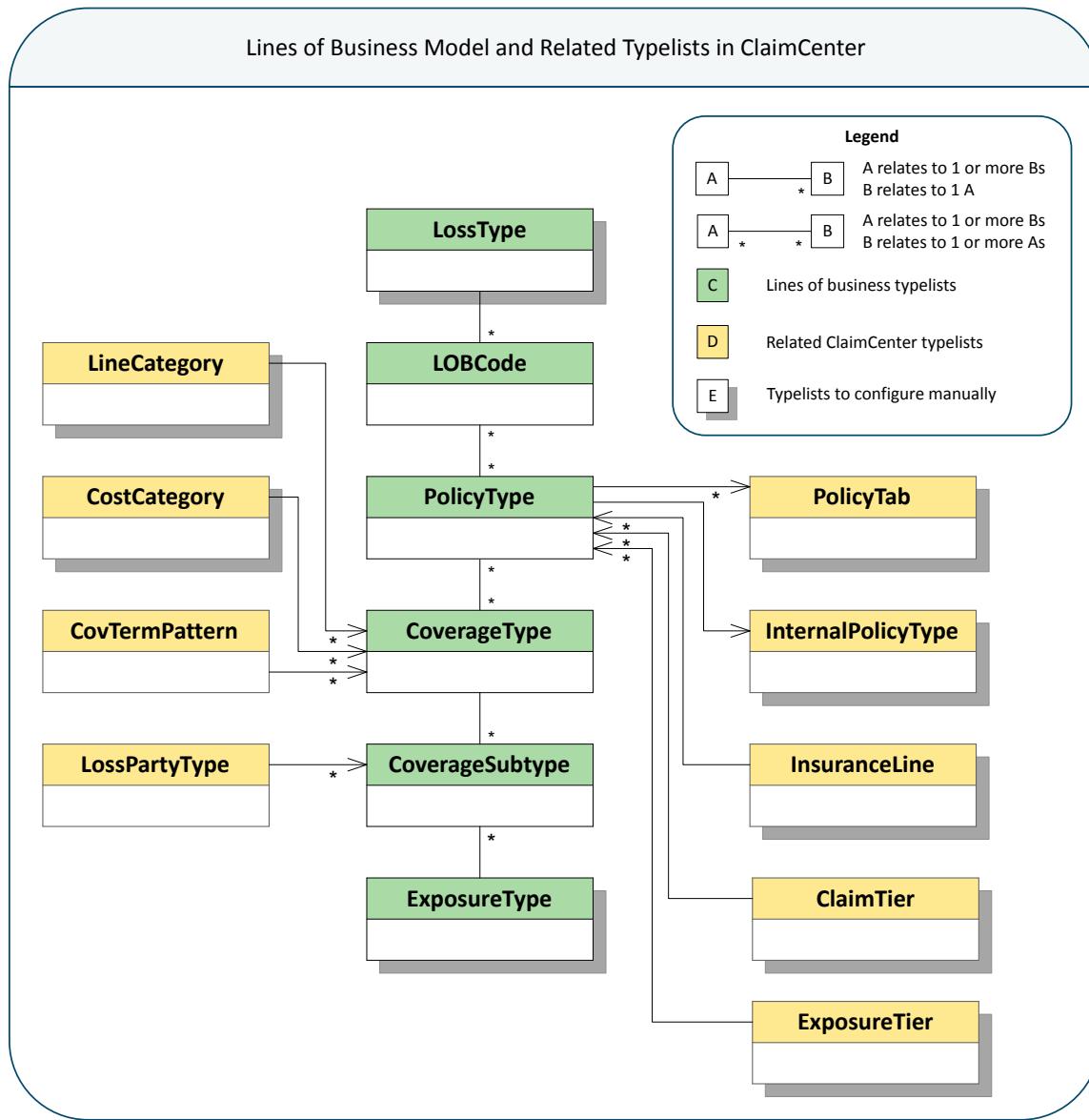
Next steps

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 492
- “Link new lines of business types to loss types” on page 492
- “Link new coverage types to an exposure type” on page 493
- “Adding new lines of business codes to related ClaimCenter typelists” on page 494
- “Adding references to codes in Gosu classes and configuration files” on page 495

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 489
- “Copy generated files to ClaimCenter” on page 492

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**.

Result

The new line of business is linked to the loss type.

Next steps

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 493
- “Adding new lines of business codes to related ClaimCenter typelists” on page 494
- “Adding references to codes in Gosu classes and configuration files” on page 495

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 489
- “Copy generated files to ClaimCenter” on page 492
- “Link new lines of business types to loss types” on page 492

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.

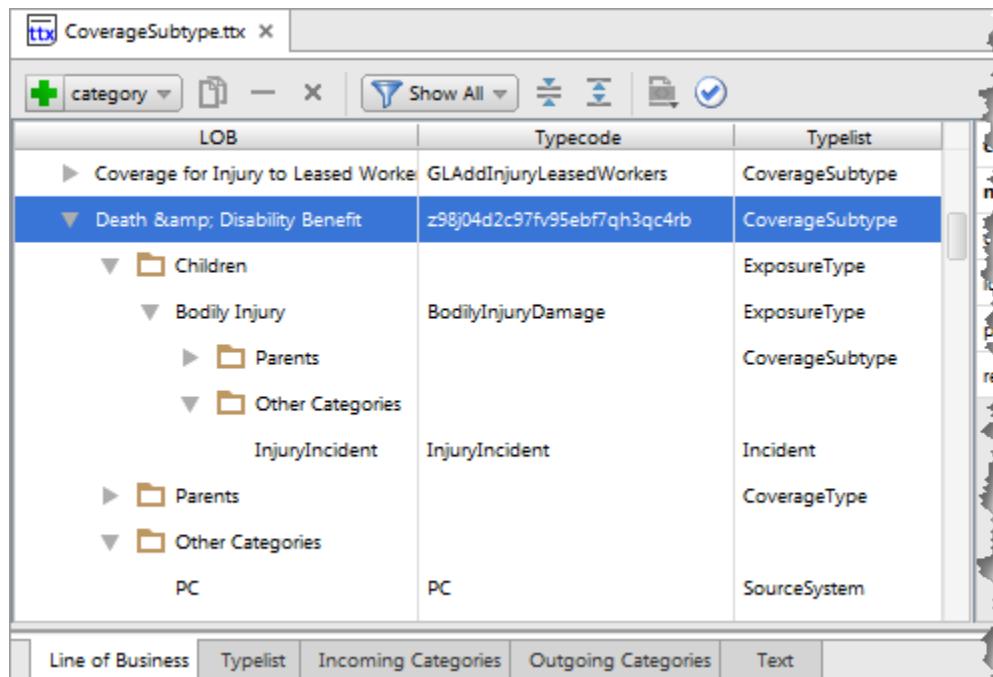
- a. Enter a value for **Code** and **Name**.
- b. Optionally, enter a value for **Description** and **Priority**.
- c. Select an **Incident**.
- d. Click **OK**.

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**.

Result

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.



LOB	Typecode	Typelist
Coverage for Injury to Leased Workers	GLAddInjuryLeasedWorkers	CoverageSubtype
Death & Disability Benefit	z38j04d2c97fv95ebf7qh3qc4rb	CoverageSubtype
└ Children		ExposureType
└ Bodily Injury	BodilyInjuryDamage	ExposureType
└ Parents		CoverageSubtype
└ Other Categories		
└ InjuryIncident	InjuryIncident	Incident
└ Parents		CoverageType
└ Other Categories		
PC	PC	SourceSystem

Next steps

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 494
- “Adding references to codes in Gosu classes and configuration files” on page 495

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 489
- “Copy generated files to ClaimCenter” on page 492

Policy location search API

Use the WS-I compliant `PolicyLocationSearchAPI` web service to retrieve summary information about policy locations within a rectangular geographic bounding box. The fully qualified path of the default Gosu implementation is shown below.

```
gw.webservice.pc.pcVERSION.ccintegration.PolicyLocationSearchAPI
```

PolicyCenter provides this web service for integration with ClaimCenter. ClaimCenter uses the returned policy location information to plot policy locations on the **Catastrophe Search** page.

PolicyCenter provide catastrophe sample data which you can use for testing.

Dependencies of the policy location search API

To return results, the `PolicyLocationSearchAPI` web service relies on the following conditions.

- The geocoding feature must be enabled.
- The `BatchGeocode` property on the relevant policy location `Address` must be set to `true`.

Finding policy locations within geographic bounding boxes

The `PolicyLocationSearchAPI` web service has a single method.

```
function findPolicyLocationByEffDateAndProductsWithinBoundingBox(effDate : Date,
    productCodes : String[], topLeftLat : BigDecimal, topLeftLong : BigDecimal,
    bottomRightLat : BigDecimal, bottomRightLong : BigDecimal) : PolicyLocationInfo[] {
```

The method accepts the following required parameters.

- The effective date for policies to find
- A non-empty `String` array of product codes for policies to find
- The bounding box coordinates for policy locations to find on policies that match the effective date and product codes. The coordinates are listed below.
 - Top left latitude
 - Top left longitude
 - Bottom right latitude
 - Bottom right longitude

The bounding box parameters are of the type `BigDecimal` with five decimal points of precision.

The method returns an array of `PolicyLocationInfo` objects.

Bounding box considerations

Keep the following in mind when specifying the coordinates of a bounding box.

- Positive latitude values represent degrees north of the equator, while negative values represent degrees south.
- Positive longitude values represent degrees east of the prime meridian, while negative values represent degrees west.
- The top left latitude must be greater than the bottom right latitude.
- The top left longitude must be less than the bottom right longitude.
- The geometric chord between the top-left and bottom-right of the bounding box must be less than 1/4 of the earth’s diameter.

What a policy location info object contains

The `PolicyLocationSearchAPI` web service returns an array of `PolicyLocationInfo` objects. Each object represents a single policy location on a policy that satisfies the effective date and product code parameters and that lies within the bounding box. A `PolicyLocationInfo` object carries information about a reinsurable policy location, including its address, the policy, the insured, and total insured values for all reinsurable risks associated with the location.

Modifying the policy location search API implementation

Generally, you do not need to modify the `PolicyLocationSearchAPI` web service. However, PolicyCenter provides the implementation in Gosu so that you can modify it if you want. For example, you might extend the `PolicyLocation` entity with custom fields that you want the web service to recognize.

The default implementation uses the Gosu class `PolicyLocationBoundingBoxSearchCriteria` to encapsulate the call to the bounding box search. Modify this class if you want to use custom fields as predicates for the query.

The default implementation uses the Gosu class `PolicyLocationInfo` to encapsulate the data that the web service returns. Modify this class if you want to include custom fields in the set of information that you return to callers of the web service. If you modify this informational class with additional properties, you must make the same modifications in the ClaimCenter version of `PolicyLocationInfo`.

Part 7

Billing integrations

Billing integration

PolicyCenter supports integration with any billing system by using plugin interfaces for outgoing requests and web services for incoming requests. The PolicyCenter base configuration includes optional integration with BillingCenter.

See also

- “Overview of web services” on page 39
- “Overview of plugins” on page 133
- *BillingCenter Application Guide*
- *Application Guide*

Billing integration overview

A typical PolicyCenter implementation integrates with a billing system. PolicyCenter has support for communication with any billing system.

PolicyCenter sends the following information to billing systems:

- Billing implications of all policy transactions
- Billing implications of policy changes
- Up-front payments for jobs
- New accounts
- New producers
- Billing implications of final audits

PolicyCenter requests the following information from billing systems:

- Payment plans available at the time that PolicyCenter creates a new policy
- A summary of billing information for a policy that PolicyCenter displays within the PolicyCenter policy file and account file user interfaces
- A preview of billing information that PolicyCenter uses during the submission process
- Availability of agency bill for a producer if the producer has an agency bill plan in the billing system

PolicyCenter also responds to requests from billing systems. For example, the billing system can cancel a PolicyCenter policy due to non-payment. Billing systems submit requests to PolicyCenter through web services that PolicyCenter publishes, which use SOAP APIs.

Mechanisms for integrating PolicyCenter and a billing system

PolicyCenter uses two mechanisms to connect to a billing system application:

- **Web services for incoming requests** – PolicyCenter exposes web services for other applications to query for information or to notify PolicyCenter about important actions. The network protocol that you use does not have to be SOAP but must provide the same functionality.
- **Plugins for outgoing requests** – Plugin interfaces define a strict contract between a Guidewire application and additional code that performs a task. PolicyCenter exposes plugin interfaces that the application calls to request or provide information. For example, if you issue a new policy in PolicyCenter, PolicyCenter calls its billing system plugin to notify the billing system of the new policy. For a new policy, the plugin might send the billing system information on how to charge the insured.

PolicyCenter provides two billing plugin interfaces, which the following table describes.

Plugin Name	Interface to Implement	Description	PolicyCenter uses during bind, issue, policy change, or renewal
Billing system plugin	<code>IBillingSystemPlugin</code>	The primary mechanism for PolicyCenter to get information from a billing system or to notify the billing system of changes to a policy.	Yes. PolicyCenter calls the billing-related plugin methods in <code>IBillingSystemPlugin</code> during critical parts of the process of creating or modifying a policy in PolicyCenter. For example, for a new policy the <code>IBillingSystemPlugin</code> provides a list of billing plans to choose from in the user interface. The billing system plugin behavior is critical to how PolicyCenter works with billing information about a policy.
Billing summary plugin	<code>IBillingSummaryPlugin</code>	To support the PolicyCenter billing summary screens in the account file and policy file, implement the billing summary plugin. PolicyCenter uses the information that this plugin returns only for the billing summary screen.	No. In theory, you can remove the billing summary screen from the PolicyCenter user interface, and PolicyCenter continues to work normally in other ways. In such a case, PolicyCenter would not call this plugin.

See also

- “Asynchronous communication between PolicyCenter and billing system” on page 504
- “Implementing the billing system plugin” on page 533
- *Installation Guide*
- *Application Guide*

Integrating PolicyCenter and BillingCenter

In the base configuration of PolicyCenter and BillingCenter, the integration between the applications uses the following two mechanisms to communicate:

- **Web services for incoming requests** – Both PolicyCenter and BillingCenter expose web services for the other application to query for information or to notify about important actions.
- **Plugins for outgoing requests** – Both PolicyCenter and BillingCenter expose plugin interfaces that the application calls to request or provide information. For example, if you issue a new policy in PolicyCenter, PolicyCenter calls its billing system plugin to notify the billing system of the new policy. For a new policy, the plugin sends BillingCenter information on how to charge the insured. The base configuration of PolicyCenter provides an implementation of the billing system plugin that calls BillingCenter web services, which in response create what BillingCenter calls billing instructions.

To integrate PolicyCenter with BillingCenter, use and customize the billing system plugin implementations in the base configuration.

Integrating policy center with another billing system

PolicyCenter uses two mechanisms to connect to a billing system application that is not BillingCenter:

- **Web services for incoming requests** – If you use a billing system other than BillingCenter, your implementations of the billing system plugins communicate with your billing system by using a network protocol such as SOAP.
- **Plugins for outgoing requests** – Plugin interfaces define a strict contract between a Guidewire application and additional code that performs a task. PolicyCenter exposes plugin interfaces that the application calls to request or provide information. If you use a billing system other than BillingCenter, you must create your own implementation classes that implement the PolicyCenter billing plugin interfaces.

How billing data flows between applications

PolicyCenter provides limited access to billing information retrieved from BillingCenter. PolicyCenter provides billing information for users who work mostly in PolicyCenter or do not have access to BillingCenter. If you have a BillingCenter login id, PolicyCenter displays links to view account and policy period information in BillingCenter. PolicyCenter has multiple examples of viewing BillingCenter data. For example, you can view billing data in PolicyCenter in the **Account File** and **Policy File** in the **Billing** tabs.

In the implementation in the base configuration, to make a midterm billing change, in general you must make changes directly in BillingCenter. However, agents typically cannot log in to BillingCenter. An agent can see and edit some BillingCenter properties in PolicyCenter within **Account File**—**Billing**. In that user interface, the payment method is editable. You can switch between responsive billing, which involves sending a bill and getting a payment, to direct debit against a credit card or bank account.

You may create additional PCF pages to show billing data or allow editing. If you want to add pages, use the PCF pages in the base configuration as examples. Copy the PCF code and integration code as needed.

For requests that require information to appear quickly in PolicyCenter, make the requests to the billing system synchronously. Similarly, if it is important to tell the user whether an action succeeded or failed, make the request synchronously. The PolicyCenter base configuration uses synchronous requests for some editing requests, such as the account file payment method. However, if you just need to update information in the billing system eventually and the user does not need to see the results, use the asynchronous approach. Follow the pattern of asynchronous PolicyCenter to BillingCenter integrations to create a custom event that uses the event messaging system, and asynchronously send that information to BillingCenter.

PolicyCenter can link directly to the billing application. For example, the PolicyCenter account billing screen has a button to view billing details directly in BillingCenter.

See also

- “Asynchronous communication between PolicyCenter and billing system” on page 504

Notifying a billing system of policy changes and premiums

PolicyCenter notifies your billing system of important changes to the policy itself and also changes that affect the premium. This affects the following policy transactions: submission, policy change, cancellation, reinstatement, renewal, rewrite, final audit, and premium report. PolicyCenter calculates premium transactions by charge type, charge group, and effective dates, and then sends them to the billing system.

If you use the PolicyCenter base configuration integration to BillingCenter, the plugin notifies BillingCenter. In response, BillingCenter creates billing instructions, which are represented by subtypes of the **BillingInstruction** entity. BillingCenter has billing instruction variations for different transaction types for new policies, new policy periods, cancellation, reinstatement, and so on. For policy jobs that accept up-front payments, the billing instruction includes information for linking the payments to a transaction.

Billing instructions are a BillingCenter concept embedded deeply in the BillingCenter data model. PolicyCenter does not directly have the concept of billing instructions. PolicyCenter notifies the billing system of policy transactions by using more generalized concepts that roughly correspond to PolicyCenter transaction types.

PolicyCenter also notifies the billing system for less common changes:

- Midterm changes to period effective and expiration dates
- Midterm change to producer of service

Tracking policies term by term

Within the PolicyCenter base configuration integration to BillingCenter, both applications track policies term by term. A policy within PolicyCenter typically starts with one term, and creates additional terms with any renewal or rewrite. However, other PolicyCenter transactions, such as policy change, create a new policy period but not a new term. Within PolicyCenter, there are various IDs for a **PolicyPeriod**, such as policy number, model number, term number, and period ID.

Your billing system must match a policy period by using both of the following IDs:

- The policy number, which remains constant throughout the life cycle of a policy period
- The term number, which starts with 1 and increments by 1 for each renewal or rewrite

PolicyCenter and BillingCenter correlate a common policy term between them by a common policy number and term number. The data models of each application differ, however. In BillingCenter, a policy number and term number uniquely identify a single entity instance. In PolicyCenter, a policy number and term number often identify multiple entity instances, due to the way that PolicyCenter implements policy revisioning.

PolicyCenter properties for billing system usage

PolicyCenter initially creates producers, producer codes, accounts, policies, and policy periods. These entities contain a number of properties to initialize during creation although only BillingCenter uses these properties.

PolicyCenter creates these objects with initial values for those properties so that additional setup in BillingCenter is unnecessary for simple cases. If you want to make additional billing changes, in particular any complicated changes, you must make those changes in BillingCenter. PolicyCenter does not show those properties in the user interface after creating the objects.

For example, a producer code has a commission plan in BillingCenter. Whenever you create the producer code in PolicyCenter, you set a commission plan which PolicyCenter sends to BillingCenter as part of the producer code. After that, you must make any changes to the commission plan in BillingCenter.

The billing integration offers the following approaches to select the property value on the PolicyCenter side, depending on the situation:

- Collect nothing from the PolicyCenter user and the billing system selects a default value. PolicyCenter behaves in this way if one likely value exists.
- Ask the billing system for a list of available plans. PolicyCenter behaves in this way if the availability logic is complex, subject to change, and only the billing system typically maintains this information.

Similarly, PolicyCenter sets some properties on **Transaction** for the billing system. Before sending the transactions to the billing system, PolicyCenter sets the transaction property **Transaction.WrittenDate** to the appropriate written date. PolicyCenter calls the policy period plugin method **determineWrittenDate**.

See also

- “Policy period plugin” on page 168

Asynchronous communication between PolicyCenter and billing system

The billing integration in the PolicyCenter base configuration uses both synchronous and asynchronous communication.

Some communication must be synchronous. In some cases, PolicyCenter needs information in response to user actions. For example, PolicyCenter needs to get available payment plans from the billing system, whether it is BillingCenter or another system. PolicyCenter must call the external system synchronously to assure a responsive user interface.

In other cases, the application needs information from the other application but the user is not waiting for a response or the application does not need a response. For example, temporary integration issues must not prevent binding a policy transaction. Because of this, the integration uses asynchronous communication in non-time-critical cases. This makes policy and billing integration robust but non-blocking to the user interface. PolicyCenter implements asynchronous communication by using the system wide event and message system.

The billing system plugin creates an event message, which triggers Event Fired rules, which create a message. In another thread, a message transport sends the message to the billing system and waits for confirmation from the billing system that it received the message.

See also

- “Messaging and events” on page 309

PolicyCenter and billing system asynchronous communication flow

The following steps show the high-level flow within PolicyCenter for asynchronous communication of a new submission to a billing system.

1. A policy period binds.
2. The submission process adds CREATEPERIOD event to the database transaction.

In the file `SubmissionProcess.gs`:

```
_branch.addEvent( BillingMessageTransport.CREATEPERIOD_MSG )
```

3. As PolicyCenter tries to commit the policy period’s bundle, the Event Fired rule set fires.
4. The **Event Fired→Billing System→Policy Period→Bind Period** rule detects this event and creates an event message.
5. The event message is put into the messaging send queue in the same transaction as the change that triggered the event (the submission).
6. PolicyCenter in a separate thread pulls a message and delivers it to the `BillingMessageTransport` plugin, which is a `MessageTransport` plugin.
7. The messaging transport (in `BillingMessageTransport.gs`) calls out to the billing plugin.

```
case CREATEPERIOD_MSG:  
...  
plugin.createPolicyPeriod(policyPeriod, getTransactionId(message) + "-2")  
break
```

8. The currently registered implementation of the billing plugin (`IBillingSystemPlugin`) handles the request. PolicyCenter includes two implementations of this interface:
 - For demonstration purposes whenever a billing system is unavailable, the class `StandAloneBillingSystemPlugin` implements this interface. This implementation does nothing with the billing information and pretends that a billing system is attached.
 - For the real integration with BillingCenter, the class `BCBillingSystemPlugin` implements this interface. See the Gosu file `BCBillingSystemPlugin.gs` for this code. Wherever this documentation refers to the PolicyCenter side of the base configuration BillingCenter integration, this class and its related implementation typically implement this behavior. This class does not handle all integration tasks. For example, PolicyCenter handles some behaviors by using web services that PolicyCenter publishes.
9. In the `createPolicyPeriod` method, the plugin handles the request by calling web service (SOAP API) calls on BillingCenter:

```
override function createPolicyPeriod( period: PolicyPeriod, transactionID : String ) : String {  
    var issuePolicyInfo = new IssuePolicyInfo()  
    issuePolicyInfo.sync(period)  
    var publicId = billingAPI.issuePolicyPeriod(issuePolicyInfo, transactionID)  
  
    return publicId  
}
```

10. PolicyCenter calls its billing plugin method `createPolicyPeriod`, which would do different things for different billing systems. In the BillingCenter integration, PolicyCenter integration code constructs an

`IssuePolicyInfo` object that is specific to web services that BillingCenter publishes. These SOAP objects are not Guidewire entities. The web service implementation on the BillingCenter side transforms this SOAP-specific object into what it calls a billing instruction. BillingCenter uses these billing instructions to track charges and instructions from the policy system.

Example creating a billing instruction from a PolicyCenter request

BillingCenter uses code like the following to handle the web service request from PolicyCenter and create the billing instruction:

```
@Throws(AlreadyExecutedException, "if the SOAP request already executed")
@Throws(BadIdentifierException, "If a policy already exists with the given number")
function issuePolicyPeriod(issuePolicyInfo : IssuePolicyInfo, transactionId : String) : String {
    var publicID = tryCatch( \ -> {
        var issuance = issuePolicyInfo.toIssuance()
        var bi = BillingInstructionUtil.executeAndCommit(issuance) as Issuance
        return bi.IssuancePolicyPeriod.PublicID
    }, transactionId)
    return publicID
}
```

See also

- “Billing instructions in BillingCenter” on page 513
- “Messaging and events” on page 309

Retryability of messages between PolicyCenter and billing system

If there is an integration error, the message error is always retryable. There are no non-retryable errors in the base configuration PolicyCenter/BillingCenter integration messaging.

In some cases, an application creates an activity whenever there is an error, so you do not need to rely on event message administration user interface to see the problem. For example, if a line of business uses renewal offers, suppose the renewal is already non-renewed or otherwise not in a state that supports automatic renewal. If the billing system receives a payment and notifies PolicyCenter, a user must decide what to do. In this case, PolicyCenter creates an activity.

Exit points between PolicyCenter and billing system applications

Parts of the PolicyCenter user interface can open a separate browser window to open BillingCenter to directly view and edit billing information for an account. The PCF feature called **Exit Points** implements these separate browser windows. To set the URL for the exit point to BillingCenter, configure the configuration parameter in `config.xml` called `BillingSystemURL`. Set this to the base URL for the server, such as:

```
http://server/bc
```

If you need to integrate with billing systems other than BillingCenter and that system requires additional parameters, you can add them to the URL. If this parameter is absent or set to the empty string, the exit point buttons in the user interface are hidden. (If the entry-point URL syntax is different enough from what BillingCenter uses, modify the PCF code for the Exit Point.)

If you need to create additional Exit Points in either application, carefully consider how you want them to work before implementing them. For example, you might want BillingCenter to have a new exit point to PolicyCenter so that a BillingCenter user can view PolicyCenter policy file information. Some things to consider in your integrations:

- How do you want this exit point to appear in the user interface?
- How do you want authentication to work between the applications? Are you using a single-sign-on authentication system? Do you have completely different sets of users for each application?
- How do you want to set permissions for BillingCenter users? For example, do you want to support BillingCenter users creating and editing PolicyCenter Policy File notes? Do you want to limit users to certain note types?

For assistance designing and implementing exit points between applications and setting up permissions in each application, contact Guidewire Customer Support.

Configuring which system receives contact updates

If a contact associated with billing information changes in PolicyCenter, PolicyCenter determines what system to notify by calling the billing system plugin method `getCompatibilityMode`. If you return `7.0.0`, PolicyCenter sends contact updates to the contact system, not to the billing system. If you return `4.0.0`, PolicyCenter sends contact updates directly to the billing system.

IMPORTANT The `getCompatibilityMode` method determines whether PolicyCenter sends contact updates to the billing system or the external contact system. Choose carefully how you want to configure this method.

Using integration-specific containers for integration

The base configuration implementation of the billing system integration passes information from PolicyCenter to BillingCenter, and from BillingCenter to PolicyCenter. These applications use web services defined as Gosu code to pass information to each other. The objects passed between applications are not direct references to persisted Guidewire entities. Each application defines core APIs to pass containers that encapsulate important information.

If you modify the integration, Guidewire strongly recommends that you maintain this pattern.

For example, if you customize or write additional web services, do not expose entities directly to the API as parameters or return objects from the web service APIs. Instead, create a Gosu class to encapsulate your integration data.

This approach makes the WSDL smaller because it does not expose the whole entities tree. It also minimizes the chance that the API breaks every time you make a domain model change during development.

IMPORTANT Do not pass Guidewire entities directly between applications. Encapsulate your data in Gosu classes that your web services expose as method parameters and as return types.

Billing producers and producer codes

PolicyCenter sends messages about producers to the billing system in the following situations:

- Whenever you create a new organization in PolicyCenter, PolicyCenter sends a message to the billing system to create an equivalent producer.
- If you change a relevant property, PolicyCenter sends a message to the billing system to update the producer. In the base configuration, a PolicyCenter user can change only properties for which PolicyCenter is the system of record.
- On system startup, PolicyCenter performs special handling of the single internal producer that corresponds to the insurer. PolicyCenter sends a message to the billing system first to determine if the billing system has the producer for the internal insurer organization. PolicyCenter calls the `producerExists` method on the billing system plugin. If the billing system does not have that producer, PolicyCenter calls the billing system plugin method `createProducer` to create the producer for the internal insurer organization.

Using producer information in BillingCenter

What BillingCenter calls a producer and what PolicyCenter calls an organization are conceptually similar but not the same. A *producer* in BillingCenter refers to a brokerage or an agency, not an individual. An *organization* in PolicyCenter represents an agency that contains individual agents. BillingCenter tracks only the agency as a producer, not the individual agents.

In the base configuration integration, you create new producers and producer codes in PolicyCenter. PolicyCenter sends a message to BillingCenter, which creates equivalent producers and producer codes. BillingCenter pays the

commission to the producer who holds a particular producer code. The producer also receives notices related to agency billing.

BillingCenter uses only some properties on producers and producer codes. Specifically, it uses properties related to commission and agency bill plans. PolicyCenter sets initial values for those properties in entity instances that it creates, so that you do not have additional setup in BillingCenter for simple cases. You edit these properties in BillingCenter.

PolicyCenter propagates the following producer information to BillingCenter asynchronously:

- New producers
- New producer codes
- Changes to name and contact information on a producer
- Changes to status on a producer code

If you make a change to a [Contact](#) entity in PolicyCenter, BillingCenter updates the [PrimaryContact](#) entity, including properties on subobjects such as [Address](#). The producer record that PolicyCenter sends to the billing system is the producer of record, not the producer of service.

If you change the [Code](#), [Organization](#), or [ProducerStatus](#) property in a producer code, PolicyCenter sends a message to BillingCenter to update the producer code. In the PolicyCenter base configuration, you can only change properties for which the application is the system of record.

The following table shows how the integration populates PolicyCenter properties to BillingCenter properties. The table also shows which application is the system of record for each property.

Business property	System of record	BillingCenter ProducerCode entity properties	PolicyCenter ProducerCode entity properties
Producer code	PolicyCenter	Code	Code
Producer codes unique public ID to identify in Guidewire applications.	PolicyCenter	Producer	Organization.PublicID
Value of Active is true if ProducerStatus is Active or Limited in PolicyCenter.	PolicyCenter	Active: boolean	ProducerStatus
The list of commission plans that PolicyCenter retrieved from BillingCenter.	PolicyCenter	CommissionPlan	CommissionPlanID

See also

- [The Data Dictionary](#)

Comparison of producer codes in PolicyCenter and BillingCenter

Both BillingCenter and PolicyCenter use the [ProducerCode](#) entity to represent producer codes. However, the data model for the entity differs between the two applications.

In BillingCenter, the producer with a particular producer code determines who receives the commission or payment related to agency billing. Therefore, every producer code has an owning producer. The producer code also provides the link to the commission plan.

The important difference between the PolicyCenter and BillingCenter data model structure is that in PolicyCenter producer codes support a hierarchical arrangement. The [Parent](#) property of a producer code is a foreign key to a parent producer code. Producer codes in the hierarchy must belong to the same organization.

PolicyCenter producer to BillingCenter properties mapping

The following table shows how PolicyCenter properties map to BillingCenter properties. The table also shows which application is the system of record for each property.

Business property	System of record	PolicyCenter Organization entity properties	BillingCenter Producer entity properties
Producer name	PolicyCenter	Name	Name
Tier	PolicyCenter	Tier	Tier
Primary contact	PolicyCenter	Contact	PrimaryContact
Direct bill commission payment properties. Defaults in BillingCenter to sending a check every month on the first day of the month.	BillingCenter	Not applicable	Direct bill commission payment properties
Agency billing plan. PolicyCenter displays a list of agency bill plans retrieved from BillingCenter.	BillingCenter	AgencyBillPlan	AgencyBillPlan
Account representatives. Optional in BillingCenter. PolicyCenter does not send a value.	BillingCenter	Not applicable	AccountRep
Unique identifier for the producer within Guidewire applications.	BillingCenter	PublicID	PublicID

See also

- The *Data Dictionary*

Billing accounts

PolicyCenter shares account information with BillingCenter. There is a one-to-one mapping between a PolicyCenter account and a BillingCenter account currency group. BillingCenter supports multiple accounts in an account currency group to support billing and payments in multiple currencies. PolicyCenter uses a single account regardless of the currency that a transaction uses. BillingCenter sends invoices from the applicable account for the currency. In the base configuration, BillingCenter uses a separate invoice stream and designated unapplied fund for each policy. Communications from PolicyCenter to BillingCenter use the BillingCenter account that has the necessary currency for the transaction.

PolicyCenter sends messages to create accounts in BillingCenter whenever the first policy for an account binds. PolicyCenter sends a message to BillingCenter to update the account whenever the following occur:

- A change to the account holder
- An addition or update to a billing contact

The native structure of the **Account** entity differs between the PolicyCenter and BillingCenter applications. Some properties on accounts are used only by BillingCenter. These properties are related to billing plan and other account-level settings for invoicing. PolicyCenter sets initial values for those properties in entity instances that it creates, so that you do not have additional setup in BillingCenter for simple cases. You must edit these properties in BillingCenter if you need to change them later.

PolicyCenter propagates the following account information to BillingCenter asynchronously:

- New accounts, but only if the first policy for that account is bound
- Changes to an account holder
- Adding a billing contact
- Changes to contact information for a billing contact

PolicyCenter does not access other accounts that your billing system might store.

PolicyCenter billing account to BillingCenter properties mapping

The following table shows the mapping between account properties in PolicyCenter and BillingCenter. In some cases, BillingCenter sets properties to default values. You can edit these properties in BillingCenter. The table also shows which application is the system of record for each property.

Business property	System of record	BillingCenter Account entity properties	PolicyCenter Account entity properties
The account number that serves as the unique ID for mapping between the applications.	PolicyCenter	AccountNumber	AccountNumber
Account name. Policy Center uses different properties on the account holder entity for the name of a person or a company.	PolicyCenter	AccountName	For an account holder that is a company, AccountHolderContact.Name. For a person, AccountHolderContact.FirstName and AccountHolderContact.LastName.
Service tier for the account.	PolicyCenter	ServiceTier	ServiceTier
Billing plan for the account. BillingCenter has default values.	BillingCenter	BillingPlan	Not applicable
Meaning of the bill date. Optional in BillingCenter. Defaults to DueDateBilling.	BillingCenter	BillDateOrDueDateBilling	Not applicable
Invoice frequency, basis, and due day. Optional in BillingCenter. Defaults to monthly invoicing with a due day of 15.	BillingCenter	InvoiceDayOfMonth, InvoiceDayOfWeek, FirstTwicePerMonthInvoiceDayOfMonth, SecondTwicePerMonthInvoiceDayOfMonth	Not applicable
Billing level for the account.	BillingCenter	BillingLevel	Not applicable
Delinquency plan. Defaults in BillingCenter.	BillingCenter	DelinquencyPlan	Not applicable
Invoice delivery types. Optional in BillingCenter. Defaults to Mail.	BillingCenter	InvoiceDeliveryType	Not applicable
Payment method. Optional in BillingCenter.	BillingCenter	DefaultPaymentInstrument.PaymentMethod	Not applicable
Primary payer. If BillingContact exists, use BillingContact. Otherwise, use AccountHolder.	BillingCenter	From the Contacts array, the AccountContact that has its PrimaryPayer property set to true	In priority order, from the AccountContacts array, the AccountContact that has in its Roles array an AccountContactRole with a subtype of: <ul style="list-style-type: none"> • BillingContact • AccountHolder
Account holder. In the base configuration of PolicyCenter, each account has only one account holder.	PolicyCenter	From the Contacts array, the AccountContact that has in its Roles array an AccountContactRole with the Role property set to AccountRole.Insured	From the AccountContacts array, the AccountContact that has in its Roles array an AccountContactRole with a subtype of AccountHolder

Business property	System of record	BillingCenter Account entity properties	PolicyCenter Account entity properties
Billing contact	BillingCenter	From the Contacts array, the AccountContact that has in its Roles array an AccountContactRole with the Role property set to AccountRole. AccountsPayable	From the AccountContacts array, the AccountContact that has in its Roles array an AccountContactRole with a subtype of BillingContact

See also

- The *Data Dictionary*

Billing plan

The billing plan contains rules about invoicing and due dates at the account level. For example, a billing plan contains all billable charges for the policies in the account that will be direct billed. The billing plan reflects the typical way that the insurer does business and is not usually an option that the producer or underwriter chooses. Therefore, PolicyCenter does not display the billing plan. If BillingCenter creates a new account as a result of the integration, the account uses the default billing plan. A BillingCenter script parameter defines the default billing plan.

The billing plan can include:

- Thresholds, such as the balance value below which an invoice is unnecessary
- Whenever payment is due
- How much charge detail to show on the invoice

The PolicyCenter base configuration does not directly use billing plans or provide an integration for this information.

See also

- “Billing Plans” in *BillingCenter Application Guide*
- *Configuration Guide*

Delinquency plan

The delinquency plan determines under which conditions the account becomes delinquent. The plan defines the normal business practices of the insurer and is not usually an option that the producer or underwriter chooses. Therefore, PolicyCenter does not display the delinquency plan. If BillingCenter creates a new account as a result of the integration, the account uses the default delinquency plan. A BillingCenter script parameter defines the default delinquency plan.

See also

- “Delinquency Plans” in *BillingCenter Application Guide*
- *Configuration Guide*

Invoicing

The frequency and timing of invoicing (due on the 25th of every month, for example) is based on the installment plan of each policy in the account. However, if the account has account-level charges and no policies in the account, BillingCenter generates an invoice based on `InvoiceDayOfMonth`. If necessary, BillingCenter uses the account defaults to create a monthly invoice stream. In the base configuration, BillingCenter uses a separate invoice stream for each policy. BillingCenter links the stream to an unapplied fund, which can be used for multiple policies. You can choose to use a new invoice stream on policy renewal.

See also

- “Billing instructions in BillingCenter” on page 513

Contacts

In PolicyCenter, each account must have one `AccountHolder` contact. Each account can have zero or more account contacts of type `BillingContact` and `AccountingContact`. There can also be one `BillingContact` on each `PolicyPeriod`. PolicyCenter sends all account contacts with `AccountHolder` and `BillingContact` roles to BillingCenter. BillingCenter sets the `Insured` role on the contact to identify the account holder contact and the `Accounts Payable` role to identify the billing contact.

On creation of a new account in BillingCenter, the primary payer is determined as follows:

- If there is an alternate billing account, use that account to determine the primary payer. The rules that follow apply to whichever of the account or alternate billing account pays the invoices.
- Because the account is being created in the context of binding a policy, if there is a `BillingContact` for the policy, set that contact as the primary payer.

Otherwise, use the following priority order to set a contact as primary payer:

- If the account has a single `BillingContact`, set this contact as the primary payer.
- If the account has more than one `BillingContact`, the first `BillingContact` that BillingCenter receives is the primary payer.
- If the account has no `BillingContact`, the `AccountHolder` is the primary payer.

After establishing the BillingCenter account, BillingCenter is the system of record for the primary payer.

PolicyCenter sends messages to BillingCenter to create new contacts and update existing contacts. Updates to BillingCenter contacts follow these rules:

- Removing a contact from an account in PolicyCenter does not trigger BillingCenter to remove the contact. The insurer’s billing department sets up contacts within BillingCenter independently from PolicyCenter, so a PolicyCenter action does not remove these contacts.
- Changing a contact in PolicyCenter to a `BillingContact` or `AccountHolder`, triggers BillingCenter to add that contact to the account.
- Changing the contact role in PolicyCenter from `BillingContact` or `AccountHolder` to another role does not trigger BillingCenter to update the role. Changing the contact back to a `BillingContact` also does not trigger BillingCenter to update the role.

Policy period merges

If you merge two accounts in PolicyCenter, PolicyCenter instructs BillingCenter to transfer all policy periods from one account to the other account. BillingCenter transfers any unbilled charges also. BillingCenter does not transfer the billing history.

Service tier

Service tiers enable an insurer to provide special handling or value-added services for certain customers, typically high-value customers. For an integration with BillingCenter, every time a submission is bound and issued in PolicyCenter, PolicyCenter sends the service tier on the account to the BillingCenter account.

The integration maps the optional `PolicyCenter.Account.ServiceTier` property to a `BillingCenter.Account.ServiceTier` property. In PolicyCenter, the typekey values for `ServiceTier` are configurable.

See also

- *Application Guide*

Billing instructions in BillingCenter

After establishing an account and policy period with billing and payment plans, the common interaction between the applications is for PolicyCenter to send charges to BillingCenter. Charges are called transactions in PolicyCenter. Transactions determine how much the account owes.

If you use the base configuration integration to BillingCenter, the plugin notifies BillingCenter of these charges. In response, BillingCenter creates billing instructions, which are represented by subtypes of the `BillingInstruction` entity. BillingCenter has billing instruction variations for different transaction types for new policies, new policy periods, cancellation, reinstatement, and so on.

Billing instructions are a BillingCenter concept reflected in the BillingCenter data model. PolicyCenter does not directly have the concept of billing instructions. PolicyCenter notifies the billing system of policy transactions by using more generalized concepts that roughly correspond to PolicyCenter transaction types.

See also

- The BillingCenter documentation

Billing instruction subtypes

The `BillingInstruction` entity is the root of several billing instruction subtypes. Multiple levels of subtypes exist, such as `AccountGeneral`, which is a subtype of `AcctBillingInstruction`, which is a subtype of the root entity. The following table summarizes the `BillingInstruction` subtypes. The arrow symbol (→) and indentation define lower subtype levels.

BillingCenter <code>BillingInstruction</code> subtype	Description and important properties
<code>AcctBillingInstruction</code>	Root for account-related billing instruction. The <code>Account</code> property of this instruction must be specified and reference the <code>PublicId</code> of the relevant account.
→ <code>AccountGeneral</code>	Specifies the effective date of the <code>AcctBillingInstruction</code> . If the <code>AccountGeneral</code> instruction is used, the <code>BillingInstructionDate</code> property must specify the effective date.
→ <code>CltlBillingInstruction</code>	Root for collateral-related billing instructions.
→ <code>CollateralBI</code>	Specifies the collateral requirement that generated the associated charge. Note: In the base configuration, PolicyCenter does not trigger code in BillingCenter that creates this billing instruction. Instead, PolicyCenter sends deposit requirements with the policy job. In other words, the <code>DepositRequirement</code> property within the main policy billing instruction contains the information.
→ <code>SegregatedCollReqBI</code>	Specifies the segregated collateral requirement that owns this charge.
<code>PlcyBillingInstruction</code>	Root for policy-related billing instruction. Includes the following additional properties. <ul style="list-style-type: none">• <code>DepositRequirement</code> – nonnegative money that specifies the deposit requirement that exists after execution of the billing instruction.• <code>OfferNumber</code> – String that contains the job number of the job that acts on the policy period to create this billing instruction.• An array of <code>PaymentPlanModifier</code> objects to apply to the existing payment plan.
→ <code>BaseGeneral</code>	Specifies the policy period and the effective date of the billing instruction. Both properties are required. The <code>Policy</code> property must specify the <code>PublicId</code> of the <code>PolicyPeriod</code> .
→ <code>ExistingPlcyPeriodBI</code>	Lets you associate the charges with an existing policy period and provide an effective date (<code>ModificationDate</code> in BillingCenter) for the charges.
→ <code>Audit</code>	Used to make changes to an existing policy as part of an audit. Properties specify whether this is a final audit and whether the charge represents the total or incremental premium for the policy. BillingCenter does not support total premium in its default integration, but it can be customized to do so.

BillingCenter BillingInstruction subtype	Description and important properties
→ Cancellation	Policy cancellation. Includes the following additional properties. <ul style="list-style-type: none"> • CancellationReason – String that describes the reason for the cancellation. • CancellationType – Values are defined in the CancellationType typelist, which has Flat, Prorata, and Shortrate in the base configuration. • HoldUnbilledPremiumCharges – Boolean field specifying whether all unbilled premium charges on the policy will be held at the end of the cancellation.
→ General	A “catch-all” general-purpose billing instruction for an existing policy. Policy administration systems may use this subtype for billing instructions that do not fit any of the other subtypes.
→ PolicyChange	Used to specify changes to an existing policy.
→ PremiumReportBI	Used to send premium report information.
→ Reinstatement	Used to reinstate a policy.
→ NewPlcyPeriodBI	All billing instructions for policy transactions that create a new policy period. This subtype has properties with a list of producer codes for the new period.
→ Issuance	Attach a new PolicyPeriod and link it to an existing Account. Automatically creates the new Policy for that PolicyPeriod.
→ NewRenewal	Lets you attach a new PolicyPeriod. It also has a link to an Account. The difference between this billing instruction and the Renewal billing instruction is that this one is for a policy that is a renewal for the insurer but new for BillingCenter. This instruction also creates a new Policy entity for the policy period.
→ Renewal	Lets you attach a new PolicyPeriod and link it to an existing Account. It expects to have a prior policy period. It links the new period to the existing policy.
→ Rewrite	Rewrite a policy. In the built-in integration, the Rewrite instruction duplicates the behavior of the Renewal billing instruction. Insurers who required specialized policy-period behavior can customize this instruction.
ReversalBillingInstruction	Reserved for Guidewire internal use.

At the time that PolicyCenter binds a job that can generate premium transactions, PolicyCenter sends a billing instruction to BillingCenter. Even if there are no charges, BillingCenter may need to know about a change to the policy period, such as knowing whether the period was canceled or reinstated. The following table summarizes what type of billing instruction PolicyCenter sends for different situations. A factor that determines what to send is whether PolicyCenter created a new policy and/or new period.

PolicyCenter Policy transaction	BillingCenter BillingInstruction subtype	Conditions and comments
submission	Issuance	Even if bind only (bind and bill, with delayed issuance), PolicyCenter generates charges for billing.
issuance	PolicyChange	PolicyCenter sends this billing instruction if the issuance creates at least one premium transaction or the following properties change: PeriodStart, PeriodEnd, or Producer Code of Record. Although PolicyCenter calls this an Issuance, because the submission already establishes the period, then PolicyCenter simply sends adjusting charges for an existing period.
policy change	PolicyChange	PolicyCenter sends this billing instruction if there is at least one premium transaction or one of the following properties change: PeriodStart, PeriodEnd, or Producer Code of Record.

PolicyCenter Policy transaction	BillingCenter BillingInstruction subtype	Conditions and comments
cancellation	Cancellation	All cancellations.
reinstatement	Reinstatement	All reinstatements.
renewal (regular)	Renewal	For regular renewals. Policy must already exist in BillingCenter. Not used in a <i>conversion on renewal</i> scenario.
renewal (new renewal)	NewRenewal	For new renewals. This is the policy period in PolicyCenter. Typically you need to create the policy in BillingCenter and create a new period. Used in a <i>conversion on renewal</i> scenario.
rewrite	Rewrite	A rewrite establishes a new period for the same policy.
final audit	Audit	All final audits.
premium report	PremiumReportBI	All premium reports.

See also

- “Billing instructions” in *BillingCenter Application Guide*

PolicyCenter financials to BillingCenter charge properties mapping

The base configuration BillingCenter integration uses the mapping of Charge properties between PolicyCenter and BillingCenter shown in the following table.

PolicyCenter property	BillingCenter property	Description
Sum(Transaction.Amount)	Charge.Amount	Transaction amount
Cost.ChargeGroup	Charge.ChargeGroup	Group together charges that are of the same charge pattern (for treatment) and can display together or be netted out only among items in the same group. This is a text property allowing arbitrary choices for grouping. Likely uses include grouping by Location or grouping charges for the same car.
Cost.ChargePattern	Charge.ChargePattern	You can choose your desired granularity of charge patterns. You can tag individual costs (premiums or taxes/surcharges) with the proper charge pattern as part of rating, if desired.
Job.JobNumber	(Charge.BillingInstruction as PlcyBillingInstruction) .OfferNumber	Identifier for use in allocating suspense payments from a billing instruction to a policy period. This property is available on the billing instruction, policy period, and suspense payment. The Charge entity does not have this property.
Transaction .EffectiveDate	Charge.EffectiveDate	Transaction effective date, useful to calculate earned premiums or equity date. In BillingCenter this is a derived property. If there is a policy change, the billing instruction has a modification date which becomes the effective date in BillingCenter.

PolicyCenter property	BillingCenter property	Description
		If the billing instruction has no modification date, as in a submission, the policy effective date becomes the charge effective date.
Transaction .ExpirationDate	Charge.ExpirationDate	Transaction expiration date, useful to calculate earned premiums or equity date.

Sending PolicyCenter transaction information to BillingCenter

When PolicyCenter rolls transactions into BillingCenter charges, PolicyCenter sends the following properties:

- Cost.ChargeGroup
- Cost.ChargePattern
- EffectiveDate
- ExpirationDate
- Job.JobNumber

Before sending the transactions to the billing system, PolicyCenter sets the property `Transaction.WrittenDate` to the appropriate written date. PolicyCenter calls the policy period plugin method `determineWrittenDate`.

See also

- “Policy period plugin” on page 168

Billing flow for new-period jobs

PolicyCenter creates a new policy upon submission, rewrite, and renewal. Whenever you create a new policy period, PolicyCenter sends a message to the billing system to create an equivalent policy period. After quoting and before binding the policy, you must select billing and payment methods. The integration retrieves billing and payment methods from the billing system. PolicyCenter displays these on the **Payment** screen. After you select a payment method, you can also preview payments. PolicyCenter retrieves the preview from the billing system to show an accurate representation of all payments and fees.

Flow of submission, renewal, and rewrite

The following series of actions and messaging occur between the applications when you quote a policy for the following jobs: submission, renewal, and rewrite.

PolicyCenter action	Messaging	Billing system action
<ul style="list-style-type: none"> • User quotes. • User advances to Payment screen. 	<ul style="list-style-type: none"> • PolicyCenter sends message to get billing options, installment plans, and invoicing plans. 	<ul style="list-style-type: none"> • Check to see if the producer code of record allows agency bill for this producer. • Look up installment and invoicing plans.
	<ul style="list-style-type: none"> • Billing system returns billing options, available installment plans, and invoicing plans. 	

PolicyCenter action	Messaging	Billing system action
<ul style="list-style-type: none"> User selects billing method. User selects installment plan. (Optional) User clicks to preview payments that billing system returned. 	<ul style="list-style-type: none"> PolicyCenter sends message with the selected payment plan. 	<ul style="list-style-type: none"> Calculate payment schedule.
	<ul style="list-style-type: none"> Billing system returns payment schedule which can be viewed in PolicyCenter. 	
<ul style="list-style-type: none"> User selects invoicing plan. (Optional) User enters one or more up-front payments. User binds policy. 	<ul style="list-style-type: none"> If the billing system does not know about the account, PolicyCenter sends account information. If the user adds an invoicing plan, PolicyCenter sends new invoicing information. 	<ul style="list-style-type: none"> If account is unknown, create account. If new invoicing plan, add to invoicing plans.
	<ul style="list-style-type: none"> PolicyCenter sends message to create a new policy and/or policy period in billing system. In that same message, PolicyCenter sends billing charges related to that policy transaction. 	<ul style="list-style-type: none"> Create policy or policy period on the account. Set <code>PolicyPeriod.BoundDate</code> to the policy's model date (the date when the policy was bound) as reported by <code>PolicyCenter.SetPolicyPeriod.TermConfirmed</code> to <code>false</code>. The policy term/period is bound, but not confirmed until payment is received and any other conditions for official binding are met. Process and apply the new charges. Process payment when received. Call the <code>IPolicyPeriod</code> method <code>hasReceivedSufficientPaymentToConfirmPolicyPeriod</code> to determine whether payment is sufficient. Payment may be sufficient if it is equal to or greater than the amount due on the renewal's first invoice. The call returns <code>true</code> if the payment is sufficient.

PolicyCenter action	Messaging	Billing system action
	<ul style="list-style-type: none"> If PolicyPeriod.ConfirmationNotificationState == NotifyUponSufficientPayment, BillingCenter notifies PolicyCenter that sufficient payment has been received by calling the PolicyCenter PolicyRenewalAPI.confirmTerm method. 	
<ul style="list-style-type: none"> Set Policy.PolicyTerm flag to true. 	<ul style="list-style-type: none"> PolicyCenter sends BillingCenter a message that the policy term/period is officially confirmed. 	<ul style="list-style-type: none"> Record the confirmation by setting PolicyPeriod.TermConfirmed to true.

Policy period mapping

The following table shows the mapping between policy period properties in PolicyCenter and BillingCenter. The table also shows which application is the system of record for each property.

Business property	System of record	PolicyCenter property	BillingCenter property
Policy account number	PolicyCenter	PolicyPeriod.Policy.Account.AccountNumber	PolicyPeriod.Policy.Account
Uniquely identify the policy	PolicyCenter	PolicyPeriod.Policy.PublicID	PolicyPeriod.Policy
Policy number	PolicyCenter	PolicyPeriod.PolicyNumber	PolicyPeriod.Policy.PCPublicID (for PolicyCenter use only) PolicyPeriod.PolicyNumber (for external PAS use)
Uniquely identifies the policy period	PolicyCenter	PolicyPeriod.PublicID	PolicyPeriod.TermNumber
Model number	PolicyCenter	PolicyPeriod.ModelNumber	PolicyPeriod.ModNumber
Period start	PolicyCenter	PolicyPeriod.PeriodStart	PolicyPeriod.PolicyPerEffDate
Period end	PolicyCenter	PolicyPeriod.PeriodEnd	PolicyPeriod.PolicyPerExpirDate
Model date	PolicyCenter	PolicyPeriod.ModelDate	PolicyPeriod.BoundDate
Product (LOB) code	PolicyCenter	Policy.ProductCode	Policy.LOBCode
Assigned risk	PolicyCenter	PolicyPeriod.AssignedRisk	PolicyPeriod.AssignedRisk
Base state	PolicyCenter	PolicyPeriod.BaseState	PolicyPeriod.RiskState
Underwriting company	PolicyCenter	PolicyPeriod.UWCompany.Code	PolicyPeriod.UWCompany
Final audit options	PolicyCenter	PolicyPeriod.FinalAuditOption	PolicyPeriod.UnderAudit
Payment plan	BillingCenter	PolicyPeriod.PaymentPlanID	PolicyPeriod.PaymentPlan

Business property	System of record	PolicyCenter property	BillingCenter property	
Is this an agency bill policy period	BillingCenter	PolicyPeriod.BillingMethod = AgencyBill	Value	PolicyPeriod.AgencyBill
Primary insured	PolicyCenter	PolicyPeriod.PrimaryNamedInsured		PolicyPeriod.PrimaryInsured
Primary producer code	PolicyCenter	PolicyPeriod.ProducerCodeOfRecord	PolicyPeriod.	PrimaryPolicyProducerCode

Billing methods and payment plans

The billing method can be agency bill, direct bill, or list bill. PolicyCenter sends a message to the billing system to determine whether the producer code of record for the producer allows that type of billing.

The billing system returns a list of payment plans available for this type of policy and account. The [Payment](#) page displays this information.

The base configuration BillingCenter integration supports both of these features.

New periods and term confirmed flag

PolicyCenter can optionally dispatch renewals to the billing system without prior confirmation from the insured. To track whether the insured confirmed the new term, PolicyCenter tracks this information in `period.PolicyTerm.Bound`. Despite the name, that property does not correspond to the policy period being bound. Instead, the property indicates whether the insured confirmed the new term. If you use this feature, you may want to send this information to the billing system while creating new terms in the billing system.

If you use BillingCenter, the base configuration billing system plugin sends this information to the billing system in the billing instruction `TermConfirmed` property.

If this property changes in PolicyCenter again, PolicyCenter calls the billing system plugin method `updatePolicyPeriodTermConfirmed`. If you use BillingCenter, the base configuration plugin implementation updates this information in BillingCenter.

If the billing system needs to later change the setting in PolicyCenter, PolicyCenter publishes a web service that an external system can call. External systems can call the `confirmTerm` method in the `PolicyRenewalAPI` web service.

See also

- “Billing implications of renewals or rewrites” on page 521
- “Policy renewal web services” on page 126

Billing flow for existing-period jobs

The following series of actions and messaging occur for the applications when you quote a policy for job types issuance (not submission), policy change, cancellation, and reinstatement.

PolicyCenter	Messaging	Billing system
• User quotes in PolicyCenter		
• User advances to Payment screen. This step applies to most jobs, but not Cancellation. A payment through the payment gateway can also be made for changes and other transactions.		
• User binds policy.	• PolicyCenter sends message to update the policy and/or policy	

PolicyCenter	Messaging	Billing system
	period in the billing system. In the same message, PolicyCenter sends billing charges related to that job.	<ul style="list-style-type: none"> • Billing system actually processes and applies new charges.

Billing implications of midterm changes

In the base configuration integration with BillingCenter, PolicyCenter handles certain types of midterm changes specially, because they do not correspond to BillingCenter billing instructions. PolicyCenter must send special updates to BillingCenter with this information.

The following topics list some of the changes that can happen in PolicyCenter. These topics do not describe the full set of billing information, some of which must directly change in the billing system in the base configuration:

See also

- “Billing flow for existing-period jobs” on page 519

Midterm changes to a policy

PolicyCenter sends BillingCenter a notice of a midterm change if the change updates specific properties on `PolicyPeriod`, summarized in the following table. The table shows which application is the system of record for each property.

Business property	System of record	PolicyCenter property	BillingCenter property
Period start (for Issuance jobs only)	PolicyCenter	<code>PeriodStart</code>	<code>PolicyPerEffDate</code>
Period end	PolicyCenter	<code>PeriodEnd</code>	<code>PolicyPerExpirDate</code>
Base states	PolicyCenter	<code>BaseState</code>	<code>RiskState</code>
Primary named insured	PolicyCenter	<code>PrimaryNamedInsured</code>	<code>PrimaryInsured</code>
Requires final audit	PolicyCenter	<code>FinalAuditOption</code> typelist: rules, yes, no	<code>RequireFinalAudit</code> Boolean: true or false
Primary producer code	PolicyCenter	<code>Producer</code>	<code>PrimaryPolicyProducerCode</code>

Midterm changes to billing method or payment plan

See also

- “Billing integration overview” on page 501

Holding billing on midterm policy transaction charges

Holding billing means for PolicyCenter to send charges to the billing system but tell it to hold further billing pending an audit.

In the base configuration BillingCenter integration, PolicyCenter only holds billing a cancellation for which a final audit is either scheduled or in-progress.

PolicyCenter continues to send premium transactions to BillingCenter for jobs as they bind/complete. Any actual holding occurs in BillingCenter itself, and does not suppress other actions within PolicyCenter.

A billing hold releases in BillingCenter if any of the following happens:

- Audit is complete. In response to PolicyCenter integration, BillingCenter creates an audit billing instruction.
- The policy is reinstated by a reversal of a cancellation.
- The policy is canceled again as a flat cancellation, which is a cancellation from the beginning of the period and with a full refund. In this case, there is no final audit because the full refund is automatic.
- A PolicyCenter user waives the final audit. Premium reporting policies disallow waiving a final audit.

Midterm changes to producer of record or producer of service

BillingCenter allows midterm change of the primary producer, which in PolicyCenter is known as the *producer of record*. PolicyCenter does not allow midterm changes to the producer of record, but can add a producer of service midterm. In the base configuration, this new producer becomes the producer of record on renewal. The new producer is the policy's representative. PolicyCenter must determine the share of commissions that go to:

- The new producer, who is representing the insured currently
- The former producer, who actually arranged the original contract

To avoid getting commissions wrong in downstream systems, PolicyCenter requires a cancel and rewrite if you want to give commission credit to the new producer. A cancel and rewrite makes the new producer the producer of record without waiting for renewal.

PolicyCenter sets the producer of service, which you can change in BillingCenter to give the new producer commissions credit. BillingCenter allows you to allocate commissions between the old and new producer according to rules for splitting the existing charges.

If in PolicyCenter, the Producer of Service changes midterm, PolicyCenter does not push an update to BillingCenter. When Producer of Service becomes Producer of Record on renewal, PolicyCenter pushes that producer to BillingCenter with the new policy period.

To give some commission credit to the new producer midterm, you must make the change in BillingCenter.

The most straightforward way to change the producer of record is to issue a cancel and rewrite in PolicyCenter. That change sends the new producer to BillingCenter as part of the new policy period.

If you must change the producer of record without a rewrite, you must perform additional customization so that PolicyCenter has the following behavior:

- Allows editing of producer of record in a policy change.
- Sends the producer change to BillingCenter.
- Provides a means to determine the midterm commission treatment for that change. The commission either has a default value or requires a change in the PolicyCenter user interface that requests the value from the underwriter. The base configuration implementation does not support this feature.

Moving a policy to a new account in midterm

In PolicyCenter, there are two ways to change the account to which the policy belongs:

- Move a policy from one account to another.
- Merge an account into another account, including moving all of the policies.

Whenever either action occurs, BillingCenter updates the account for that policy.

Billing implications of renewals or rewrites

PolicyCenter handles multiple issues related to billing for renewals and rewrites.

Choosing the renewal flow type

PolicyCenter provides multiple renewal flows from which you can choose. Use only one of these renewal flows. The following table compares and contrasts the different renewal flows, with the first column containing the general name for this renewal flow.

Re-newal flow	How PolicyCenter renews a policy	If the insured pays	If the insured does not pay
Bind and cancel	<p>In the typical case, renewal occurs without user intervention. First, PolicyCenter quotes the renewal. Next, PolicyCenter sets the job workflow status to Renewing. The workflow behavior after this status change is configurable, based on how close the current date is to the effective date of the renewal. If the renewal effective date is sufficiently in the future, the workflow waits. After a configurable date, such as 30 days before the renewal effective date, the workflow automatically issues the renewal.</p> <p>On issuance of the policy renewal:</p> <ul style="list-style-type: none"> • PolicyCenter sends charges to BillingCenter. • The <code>PolicyTerm.Bound</code> property becomes true. <p>Before PolicyCenter automatically sets the workflow status to Renewing, you can manually make this change in the user interface. Within the policy renewal job, click Edit, make changes, click Bind Options, and then click Renew. The job workflow status becomes Renewing.</p> <p>Before PolicyCenter automatically issues the policy, you can manually make this change in the user interface. Within the policy renewal job, click Edit, make changes, click Issue Now. The renewal immediately issues.</p>	<p>By this time, the policy is already issued. No special action happens.</p>	<p>By this time, the policy is already issued. If you use BillingCenter, and BillingCenter does not receive a payment, BillingCenter instructs PolicyCenter to perform a “cancel for non-payment”. The delinquency plan determines whether BillingCenter specifies a flat cancel or midterm cancel.</p> <p>After BillingCenter requests cancellation of the policy, any money received after any cancellation remains in the account unapplied fund.</p>
Re-newal offer	<p>In the typical case, renewal occurs without user intervention. First, PolicyCenter quotes the renewal. Next, PolicyCenter sets the job workflow status to Renewing, which is a waiting state.</p> <p>When the job workflow status changes to Renewing, you must configure your Event Fired rules to send an offer to the insured:</p> <ul style="list-style-type: none"> • The base configuration does not provide an integration to send an offer to the insured. You can decide whether to provide the insured with a single payment plan or multiple options. To get the payment plans, configure PolicyCenter to request this information from the billing system. Include any 	<p>The billing system sends PolicyCenter notice of payment by using the PolicyCenter <code>PolicyRenewalAPI</code> web service method <code>notifyPaymentReceivedForRenewalOffer</code>.</p> <p>If you use BillingCenter, this happens because BillingCenter registers the payment including the offer reference number. BillingCenter holds the payment as a suspense payment with an associated offer reference number. A suspense payment is a payment not yet matched to an account and a policy.</p>	<p>If the workflow status is Renewing, the job waits for the insured to respond to the offer.</p> <p>If the insured did not respond by the renewal deadline, the workflow job status changes to Not Taken (<code>NotTaken</code>) status.</p>

Renewal flow	How PolicyCenter renews a policy	If the insured pays	If the insured does not pay
Automatic renewal	<p>payment plan information on a renewal offer letter.</p> <ul style="list-style-type: none"> The offer must contain a reference number to identify this job. Use the renewal job ID as the PolicyCenter offer identifier. <p>In some cases, the automatic renewal does not reach the Renewing status, for example, if the policy had approval problems. In the policy renewal job, click Quote. The renewal offer status changes to Renewing. This status change causes your Event Fired rules to send the offer letter to the insured.</p> <p>At this point, the property <code>PolicyTerm.Bound</code> is false.</p>		<p>If the workflow status changed to Not Taken and money arrives afterward, the billing system tells PolicyCenter about the payment received. PolicyCenter does not accept this request. Instead, PolicyCenter returns an error to the billing system that the renewal is incapable of renewal. If you use BillingCenter, BillingCenter creates an activity for someone to look into the new payment that it cannot apply.</p>
Confirmed renewal	<p>The confirmed renewals flow can be summarized as follows:</p> <ul style="list-style-type: none"> PolicyCenter binds and issues the renewal. PolicyCenter waits for confirmation from the insured that the insured wants the policy. PolicyCenter sets the property <code>PolicyTerm.Bound</code> to <code>false</code> until receiving sufficient payment. 	<p>After receiving sufficient payment for the renewal, the billing system notifies PolicyCenter. For example, the billing system calls the PolicyCenter <code>PolicyRenewalAPI</code> web service method <code>confirmTerm</code>.</p> <p>The <code>confirmTerm</code> method performs the following actions:</p> <ul style="list-style-type: none"> PolicyCenter sets <code>Policy.PolicyTerm.Bound</code> to true. PolicyCenter sends a message to the billing system to confirm the renewal. 	<p>Eventually, the billing system notifies PolicyCenter of a Not Taken cancellation. All Not Taken cancellations are flat cancellations.</p>

The following table lists how to configure each type of renewal flow.

Renewal flow	Configuration overview	More information
Bind and cancel	The policy renewal plugin (<code>PolicyRenewalPlugin</code>) has a method called <code>doesRenewalRequireConfirmation</code> , which takes a <code>PolicyPeriod</code> object. To require confirmation, your plugin implementation of that method must return <code>true</code> . Otherwise, return <code>false</code> .	"Not taken renewals cancellation" on page 527
Renewal offer	The policy renewal plugin method <code>isRenewalOffered</code> takes a <code>PolicyPeriod</code> object. To support renewal offers, your plugin implementation of that method must return <code>true</code> . Otherwise, return <code>false</code> .	<p>"Renewal offers flows" on page 524</p> <p>"Not taken renewals cancellation" on page 527</p>
Confirmed renewal	The policy renewal plugin method <code>doesRenewalRequireConfirmation</code> takes a <code>PolicyPeriod</code> object. To support confirmed renewal, your plugin implementation of that method must return <code>true</code> . Otherwise, return <code>false</code> .	<p>"Confirmed renewals flow" on page 525</p> <p>"Not taken renewals cancellation" on page 527</p>

If you want to choose different renewal flows for different renewal jobs, add your own logic for the plugin methods described in the previous table. For any renewal job, your policy renewal plugin logic must define exactly one

renewal flow. For example, for one renewal job you cannot return true for both `doesRenewalRequireConfirmation` and `isRenewalOffered`.

In the base configuration, all lines of business use the bind-and-cancel flow.

Limitations of renewal flow types

Bind-and-cancel renewals have one main limitation:

- If the insured fails to pay, no coverage is provided, which can cause ambiguities. Imagine that the date occurs in the period in which the renewal term appears to be bound in PolicyCenter but no payment was received yet. If the insured files a claim today, the insurer must make it clear to the insured that coverage is uncertain. If the insurer receives payment within the deadline, there is coverage. Otherwise, the insurer must deny the claim. This is especially a problem if insurers provide a grace period after the start of the renewal period for receiving payment. In the bind-and-cancel renewal, there is no clear indication in the data model that the renewal period is uncertain.

Renewal offers have two main limitations:

- If PolicyCenter considers the policy issued in PolicyCenter, does that mean that the insurer must implement advanced notification and justification steps to cancel? That might depend on many factors, including jurisdiction, and also the wording to the insured about whether renewal is contingent on payment by a certain date. The fact that no renewal term yet exists because the job is unfinished complicates the question of coverage and billing
- For renewal offers, BillingCenter is not in charge of the invoicing for the renewal offer itself. Some insurers want BillingCenter in charge of the renewal offers, particularly for receiving electronic payments.

Renewal offers flows

This topic describes how the base configuration billing integration handles renewal offers.

PolicyCenter selects the proper payment plan before issuing the renewal, but PolicyCenter does not requote the policy period to recalculate premiums. BillingCenter sends PolicyCenter an amount paid and optionally a payment plan code to indicate what the insured chose. Best practice is to use an explicit plan choice, but PolicyCenter can choose the plan that seems most appropriate. If BillingCenter does not provide a plan code, PolicyCenter chooses the payment plan with the largest down payment requirement less than or equal to the amount paid. Based on the amount paid, PolicyCenter makes the best assessment of whether the insured wants to pay all up-front, or to choose 2-payments, 3-payments, and so on.

PolicyCenter verifies that the amount paid is sufficient to meet the down-payment required. If the insured does not send enough to meet the selected plan, PolicyCenter raises an error and an activity to investigate is generated for a BillingCenter user. If the insured provides no plan code, PolicyCenter checks whether the amount is sufficient for any plan. If no (adequate) payment signal is received in time, the PolicyCenter workflow moves the renewal job to not taken.

In the renewal offers flow, receiving a payment signals that the insured accepted the offer. The billing system sends PolicyCenter notice of payment from BillingCenter by using the `PolicyRenewalAPI` web service method `notifyPaymentReceivedForRenewalOffer`. This API also sets `PolicyTerm.Bound` to true. This means that `Bound` is true whenever the renewal job completes. Only then does the insurer bind the policy.

The insurer initially sends a renewal notice (including pricing and payment plans) but does not book premiums for accounting or statistics. If payment is received, the billing system tells PolicyCenter to bind. Otherwise, a PolicyCenter timeout triggers PolicyCenter to mark the renewal offer as not taken. The timeout includes any grace period in which they can reinstate with lapse.

In some cases, an insurer indicates payment options in the renewal offer, such as \$500 for full payment or \$100 down payment for an installment plan. Based on the amount received, the insurer decides which option was selected and tells PolicyCenter to bind the renewal and chooses which. The actual premiums calculated by rating will be different depending on the payment plan chosen, so PolicyCenter requotes the policy with the option chosen prior to binding the renewal.

Note: Some insurers may accept payment within a grace period. In other words, if the payment is past the grace period, the renewal adjusts to start on the date of payment, not on the original renewal period effective date. The policy renews but with a lapse in coverage. However, PolicyCenter and BillingCenter do not support automatic renewal with lapse in the base configuration.

It is also possible to receive a payment signal from BillingCenter while PolicyCenter is not in a waiting state. PolicyCenter handles the following cases:

- If PolicyCenter renewed the policy already (this is a race condition), PolicyCenter returns in a way that causes the payment to apply to the renewal without raising any errors.
- If it is any other wrong status, PolicyCenter returns an error that creates a new activity in BillingCenter to investigate. For example, the BillingCenter user might need to determine whether:
 - To return the funds
 - To redo the renewal, since the insured apparently does want a policy
 - The renewal was undergoing changes but was still issued

See also

- “Not taken renewals cancellation” on page 527
- “Confirmed renewals flow” on page 525

Customizing when to use renewal offers

In the base configuration, all lines of business use the bind-and-cancel renewal flow. No lines of business use renewal offers. You can change which flow to use for any of these lines of business. For new lines of business that you create, you can choose either renewal flow.

PolicyCenter configures the renewal flow choice by calling the `isRenewalOffered` method of the registered implementation of the policy renewal plugin interface (`IPolicyRenewalPlugin`). If this method returns `true`, PolicyCenter uses a renewal offer for this policy renewal. Otherwise, PolicyCenter uses the bind-and-cancel flow.

You can modify the base configuration implementation of this plugin or customize the `isRenewalOffered` method directly.

You can modify or override the `isRenewalOffered` method to support renewal offers for a particular line of business. For example, the following `isRenewalOffered` method supports renewal offers for commercial auto (`BusinessAuto`):

```
override function isRenewalOffered( periodToRenew : PolicyPeriod ) : boolean {  
    return periodToRenew.Job typeis Renewal  
        and periodToRenew.Policy.ProductCode == "BusinessAuto"
```

Depending on what regions you support and the laws, you might need this method to vary the result by jurisdiction. For example, in the United States, some states allow bind-and-cancel. However, some states prohibit it because they do not want insurers to bind a policy before payment.

See also

- “Not taken renewals cancellation” on page 527

Confirmed renewals flow

For some insurers, neither the bind-and-cancel nor the renewal offers flows are appropriate.

Confirmed renewals are an alternative flow that combines qualities of bind-and-cancel and renewal offer flows:

- Confirmed renewals allows the job to complete immediately, like the bind-and-cancel flow.
- BillingCenter has complete control of the billing aspects to the insured, like the bind-and-cancel flow. This is particularly valuable for handling electronic payment methods.
- PolicyCenter always knows whether the policy is fully bound or only tentatively bound thanks to an additional data model field: `Policy.PolicyTerm.Bound`. This is initially false for renewals that need confirmation. The billing system calls the PolicyCenter `PolicyRenewalAPI` web service method `confirmTerm`. This sets the `Bound`

field to `true`. This change happens even though the renewal job finished. In the base configuration, for BillingCenter users, PolicyCenter sends this information in the billing instruction `TermConfirmed` property.

- The bind-and-cancel approach starts immediately booking financials because there is no way to encode the concept of pending confirmation. Some insurers do not want to book written and earned premiums or billed receivables until a renewal is paid by the insured. In confirmed renewal, the insurer can confirm that the insured actually wants the policy before starting those processes. This flow waits until confirmation before accruing written and earned premium.

Note: The implications for premiums and receivables with confirmed renewals vary widely by insurer. Carefully consider the behavior that you require from confirmed renewals before implementation.

- For confirmed renewals, the billing system gives PolicyCenter positive confirmation whenever the insured pays the billing system. The bind-and-cancel flow does not have this information.
- For confirmed renewals, PolicyCenter knows whether the insured never confirmed the policy and that therefore the policy is not legally binding. This information is useful for handling ambiguities like those mentioned earlier in this topic. The bind-and-cancel flow does not have this information.

The base configuration of PolicyCenter does not provide waiting until confirmation. You can change this behavior.

See also

- “Renewal offers flows” on page 524
- “Billing flow for new-period jobs” on page 516

Account creation for conversion on renewal

The conversion on renewal process moves policies into PolicyCenter at renewal time. In some cases, these policies are also new to the billing system.

If the policy creates an account during a conversion on renewal, PolicyCenter sends the information to BillingCenter it first checks if the account already exists in the billing system. If it does not, PolicyCenter establishes the new account in BillingCenter just like a submission.

See also

- “Policy renewal web services” on page 126

Copying billing data to new periods on renewal or rewrite

Whenever PolicyCenter creates a new period for a renewal or a rewrite, PolicyCenter handles properties for which BillingCenter is the system of record as follows:

- If PolicyCenter must view or edit the property, PolicyCenter retrieves the latest value from BillingCenter on starting the renewal or rewrite. These properties are:
 - Agency Bill or Direct Bill, if a choice is available
 - Payment Plan
- For other properties, BillingCenter copies those values to the new periods from the prior period.

Billing implications for cancellations and reinstatements

PolicyCenter handles multiple billing issues related to cancellations and reinstatements:

See also

- “Billing flow for existing-period jobs” on page 519
- “Billing implications for premium reporting” on page 530

Cancellations that start in PolicyCenter

If PolicyCenter binds a cancellation, PolicyCenter notifies BillingCenter, which creates a cancellation billing instruction.

The following are examples of cancellations that PolicyCenter starts:

- Cancellations for the purpose of doing a rewrite.
- Cancellations at the request of the insured.
- Cancellations because the insurer has grounds for cancellation.
For example, the insured is found to have lied in their application or violated the contract.
- Cancellations for a newly bound renewal, but the insured informs the insurer that do not want the renewal.

Cancellations that start in BillingCenter

Delinquency cancellation

The primary example of delinquency cancellation is cancellation for non-payment. If there are overdue invoices, BillingCenter starts a delinquency process. As part of the delinquency process, BillingCenter tells PolicyCenter to cancel as soon as possible, ideally immediately. In the base configuration, PolicyCenter calculates the actual cancellation date by using the minimum lead time required by law.

If BillingCenter starts a cancellation, the integration performs the following steps:

1. BillingCenter sends a message to PolicyCenter to cancel.
2. PolicyCenter cancels the policy.
3. PolicyCenter sends a **Cancellation** billing instruction back to BillingCenter.
4. BillingCenter marks the policy period as canceled on receipt of the **Cancellation** billing instruction.

IMPORTANT If delinquency triggers cancellation, BillingCenter requests immediate cancellation. The base configuration integration does not support BillingCenter requesting PolicyCenter to cancel the policy as of a certain date. You can customize the integration to support specified dates for cancellation.

Not taken renewals cancellation

There are three types of renewal flows:

- Bind-and-cancel
- Renewal offers
- Confirmed renewals

For a line of business that uses the bind-and-cancel renewal flow, PolicyCenter binds a renewal policy period as part of the renewal offer process. If the user does not pay, the lack of payment in the billing system indicates that the policyholder does not want to take the renewal. Next, the billing system uses web services to initiate a flat cancel order in PolicyCenter. In this case, the cancellation reason is not taken rather than non-payment. The not taken delinquency process does not require dunning letters and threats to cancel. The handling of dunning letters differentiates the not taken reason from non-payment.

For a line of business that uses confirmed renewals, the billing system notifies PolicyCenter of the not-taken cancellation if the insured does not pay by the deadline. This causes PolicyCenter to stop referring to the policy term as pending confirmation, but instead as cancelled.

For confirmed renewals, the original renewal letter indicates that coverage is expiring but the company provides renewal coverage only if the company receives payment by a specified date. If the company does not receive the payment, BillingCenter sends PolicyCenter a message to cancel the policy with a not-taken cancellation reason. PolicyCenter flat cancels the policy as of the effective date of the policy period.

In all of these cases, neither PolicyCenter nor the billing system need to implement dunning letters and threats to cancel.

If you use BillingCenter, BillingCenter has its own workflow to manage the delinquency process for renewals. If the insured fails to pay within a specified time, BillingCenter begins a delinquency process for non-payment of a renewal.

The not-taken renewal process is as follows:

1. The PolicyCenter automated renewal process initiates a renewal, creates a quote, and binds the policy period.
2. PolicyCenter sends BillingCenter the new policy period and charges on the renewal billing instruction.
3. The controlling Delinquency Plan includes the `NotTaken` delinquency reason in the workflow. The workflow type is `CancelImmediately`. The account delinquency plan initiates if the `PolicyPeriod` does not contain a delinquency plan. If both the account and the policy period have an assigned delinquency plan, the plan for the policy period prevails.
4. BillingCenter bills the down payment as specified by the payment plan.
5. If BillingCenter does not receive payment for the initial invoice of the policy period, BillingCenter starts a flat cancellation in PolicyCenter.

See also

- “Choosing the renewal flow type” on page 521
- “Billing implications of renewals or rewrites” on page 521

Billing implications of audits

Audits affect billing in multiple ways, both before and after an audit occurs.

See also

- “Billing flow for existing-period jobs” on page 519

Holding periods open for audits

Under normal circumstances, BillingCenter closes a policy period if there are no outstanding charges, no outstanding balance, the expiration date passes, and all premium is earned. Basically, closing the period means that BillingCenter does not expect any more activity on that period.

If a policy is subject to a final audit, PolicyCenter tells BillingCenter to set the period status to `OpenLocked` while it creates the period. This prevents it from being closed before the final audit. When PolicyCenter sends the final audit billing instruction, BillingCenter sets the period back to Open. The batch process that closes periods soon closes the now-audited period.

For a policy originally subject to final audit, a PolicyCenter user can later decide to waive final audit. An insurer typically does not want the billing system to wait forever with the period in an open and locked status because it never got the final audit. Waiving the audit unlocks the policy. If you use the base configuration BillingCenter integration PolicyCenter notifies BillingCenter about waiving the audit and BillingCenter unlocks the policy period. This removes the `OpenLocked` status.

Less likely is that policy period might start out not subject to a final audit, but later the underwriter decides that an audit is appropriate. If you schedule a final audit in PolicyCenter, PolicyCenter notifies BillingCenter that the period must stay open for the audit.

Generating an audit report

A bill for an audit contains the following items that a policyholder must understand:

- The total audited premium for the period. This is basically a quote page from PolicyCenter.
- The amount due on their invoice

Insurers traditionally provide a bill that shows the value of total amount due, which is calculated from the following formula:

```
TOTAL_AMOUNT_DUE = TOTAL_AUDIT_PREMIUM - AMOUNT_PREVIOUSLY_BILLED_AND_COLLECTED
```

PolicyCenter cannot calculate the total amount due because PolicyCenter does not know the amount that the billing system already collected. PolicyCenter and BillingCenter do have the following information:

- PolicyCenter knows the difference between the audited premium and what was previously sent to billing in transactions that PolicyCenter already charged. Call this amount X.
- BillingCenter knows the value of currently uncollected amounts, including anything that might not yet have billed, if that is possible. Call this amount Y.

The total due is X+Y, just as it would be for any new charges. The package that the insurer sends to the policyholder typically includes the following:

- An invoice showing any open balance plus any new charges including audit amount X
- An audit report showing Total Audit Premiums and Change in Premiums. This is the data from the two tabs on the quote page in PolicyCenter that explains the value derived in the amount X.

An insurer can optionally add code to PolicyCenter to call out to a billing system to display difference between the final audit and the amount paid. This would not change what PolicyCenter must send to the billing system but it might amend what to print on the final audit statement.

Sending audit premiums as incremental

Like any other policy transaction, PolicyCenter must send billing charges to the billing system to bill for premium changes resulting from an audit.

There are two ways to think about the results of an audit.

- Policy system sends total audit premiums to billing. In that case, the billing system must compare this to the amount previously billed. This would include any deposit but not include various fees on top of the charges sent from the policy system. The policy system explains this to the insured and bills the insured for the difference.
- Policy system sends incremental charges compared to what was previously sent to billing system as with other transactions.

The base configuration BillingCenter integration uses the second approach. The base configuration PolicyCenter behavior sends BillingCenter incremental charges only compared to what PolicyCenter previously sent to the billing system. PolicyCenter looks at transactions from previous versions of the policy to determine what PolicyCenter already sent to the billing system.

Audit reversals and revisions

There are times that an audit must change. There are two basic situations:

- An audit revises, which causes PolicyCenter to send adjusting charges compared to the prior audit.
- The audit reverses, other changes may be made to the policy, and then a new audit happens. A variation of this is reversing the audit because the policy reinstates. There may then be many other transactions before a new audit is done months later.

For a revised audit, the PolicyCenter base configuration integration to BillingCenter creates another audit request with additional charges. BillingCenter creates an audit billing instruction to handle this request.

For an audit reversal, a few things happen:

- PolicyCenter sends a new Audit billing instruction with new charges. These reverse the prior audit. However, the total might be positive or negative, so PolicyCenter must tell BillingCenter that this is a reversal, not just a revised audit. BillingCenter cannot derive that information from the total charges.
- BillingCenter moves the policy period back to OpenLocked because the policy needs a new audit before it can finish. This can occur for a policy change after final audit or a reinstatement after final audit.

Billing implications for premium reporting

PolicyCenter handles some policies on a reporting basis. In this case, from the PolicyCenter user interface, you choose a reporting plan instead of an installment payment plan.

If you use BillingCenter, the reporting plan implies a simple payment plan in BillingCenter.

Sending reported premiums to your billing system

For a policy using premium reporting, some or all costs are subject to reporting. During regular policy jobs (submission, issuance, renewal, policy change, cancellation, reinstatement, rewrite), PolicyCenter does not send those charges to the billing system.

In the base configuration integration to BillingCenter, PolicyCenter sends the deposit requirement and costs that are not subject to reporting to BillingCenter. For examples, taxes might not be subject to reporting.

At the end of a premium report in PolicyCenter, PolicyCenter sends the premium transactions to the billing system. If you use BillingCenter, BillingCenter converts the associated web service request from PolicyCenter to a premium report billing instruction.

In addition to sending the charges to the billing system, PolicyCenter sends whether the company already received the payment. Sometimes a PolicyCenter user enters a premium report prior to posting the payment in BillingCenter. If so, BillingCenter does not send the insured an invoice for the amount due if the insurer already knows that the payment was received. Even if the payment was not paid accurately, PolicyCenter wants the billing system to wait until the payment posts. Only after that does BillingCenter determine the difference between the amount owed and the amount received. The base configuration integration to BillingCenter includes information about whether the payment was received. BillingCenter uses this information to create a `PremiumReport` billing instruction.

However, it is more common for a billing system to receive the premium report payment before the premium report notification from PolicyCenter. Checks and reports are often send together and the insurer usually deposits the check immediately before the report arrives in the audit department for data entry. If so, the billing system assigns the payment to the account as a suspense payment and waits to receive the report.

The BillingCenter part of the integration sets the Premium Report due date as the override `PaymentDueDate` on the premium report. This ensures that BillingCenter knows to expect the payment by that date rather than waiting for the typical next invoice plus 30 days date.

See also

- “Billing flow for existing-period jobs” on page 519

Billing implications of delinquency for failure to report

The billing system usually initiates workflows to handle delinquency for non-payment. In contrast, the policy system typically handles delinquency for premium reporting.

PolicyCenter tracks the audits, and therefore tracks and initiates delinquency for premium reporting. In the base configuration integration, PolicyCenter creates an activity for the underwriter to follow up with the insured.

You can optionally configure some other more complex delinquency process that include an automatic cancellation request. For example, you can add additional integrations such as the following:

- Reprint or resend the report to the policyholder with a delinquency message, such as “If this report is not received in 10 days your policy is subject to cancellation.”
- The audit item includes an additional follow-up date, which is `null` until a batch process identifies that the report is past due. At that time, you can reprint the report with the proper message.
- An activity, or some other notification, to the producer.
- Start an automatic cancellation. If the delinquency is on a direct bill premium report, this delinquency is for non-payment. Otherwise, you typically call this a non-report cancellation.

Billing implications of deposits

A deposit is not the same as a down payment. A *deposit* is collateral collected up front on a policy that bills based on reporting. A *down payment* is an initial payment of a portion of the premium. In the insurance industry, people use these terms loosely. Guidewire documentation uses these terms with their precise meanings.

PolicyCenter can receive a deposit as an up-front payment. Similarly, PolicyCenter can take an additional up-front payment as changes, renewals, and rewrites modify the amount of the deposit that the policy requires. PolicyCenter sends the information about up-front payments to BillingCenter.

The reporting plan typically specifies a deposit requirement as a percentage of the premium subject to reporting. On submission, PolicyCenter calculates the required deposit and it is billable immediately. PolicyCenter can receive this deposit as an up-front payment.

On a policy change, the required deposit may change based on the total premium changing and the deposit being a percentage of total premium. PolicyCenter shows the amount of required deposit as part of quoting the policy.

PolicyCenter determines the deposit based on total premium and the deposit percentage for the reporting plan that the PolicyCenter user chose. It sends the deposit to the billing system as an absolute number, for example a \$500 deposit for each PolicyCenter job that PolicyCenter sends to BillingCenter.

BillingCenter tracks this amount for each policy period and uses it to establish a required deposit, a bill to collect that deposit, and so forth. BillingCenter implements a default function for determining the deposit for a policy based on the deposit requirement for each policy period. Alternatively, you can configure BillingCenter to support a rolling deposit, the maximum deposit of all open periods, or a straight deposit, the sum of deposits of all open periods.

PolicyCenter segregates the collateral requirement for each policy period but stores collateral information on the account. If you change the collateral requirement, PolicyCenter establishes a new collateral requirement. If you set it to zero, PolicyCenter closes the existing collateral requirement.

Implications for deposits vary by job type.

Deposits in submission, renewal, or rewrite jobs

When the PolicyCenter user selects a reporting plan on the **Payment** screen, PolicyCenter displays a default deposit requirement. PolicyCenter gets this default value from the audit schedule that you configure in Studio.

The deposit amount is a percentage of the total cost subject to reporting. Typically, taxes are not subject to reporting. The subject-to-reporting premium is the subset of the total premium for which premium costs are marked as subject to reporting.

PolicyCenter users can override the default deposit. If you override a default deposit, PolicyCenter recalculates the deposit requirement and displays the value in the user interface.

When the PolicyCenter user binds a policy, PolicyCenter sends the deposit collateral amount to BillingCenter with the **PolicyPeriod**. In response, BillingCenter stores this data in the property

`PlcyBillingInstruction.DepositRequirement`. This deposit is the amount of collateral to consider segregated for the specified policy period.

When BillingCenter receives the deposit amount, BillingCenter creates a Collateral Requirement of type **Cash** on the BillingCenter account. This collateral requirement is segregated. In the user interface BillingCenter displays the status “not compliant” until the insured pays it. The BillingCenter property `ColleralRequirement.Compliance` controls collateral requirements.

BillingCenter also creates a collateral charge. The charge generates a collateral invoice item which appears on an invoice to the account holder. Whenever the insured pays the collateral invoice items, BillingCenter adds this to the amount on the collateral requirement. Because it is segregated, BillingCenter shows the cash balance in the category of “cash held by requirements”. If the entire amount is paid, the collateral’s compliance status (the `ComplianceStatus` property) changes to **compliant**.

Each policy period (policy term) has its own segregated collateral requirement stored on the account.

Applying deposits to a renewal period

The deposit requirement is based on the total premium for a single policy period. In theory, the billing system holds the money until the period is paid in full. Alternatively, the billing system covers any final amount due and then releases the excess. If the policy does not renew, this is what happens.

When the policy renews, insurers and insureds typically want to transfer the deposit to the renewal period. They do not want to pay an additional deposit for the renewal period if the deposit on the expiring period is still held. The renewal period overlaps with the expiring period if binding occurs before the end of the expiring period. The final audit or final premium report on the expiring period may not complete until months after the period ends. During that time, in theory, collateral is necessary for any premium due on the old period and for any premium earned on the new period. In practice, some insurers are satisfied with holding only one deposit that covers both periods. In this case, the deposit from the old period does not net against the remaining amount due. Instead, the billing system bills the full additional premium, for example from an audit, for the old period and retains the deposit for the renewal period.

Typically, the insurer eventually transfers this amount to either pay off the final audit balance or applies the funds to the segregated collateral requirement of the renewal policy period.

When the released collateral moves to the renewal period, the expiring policy usually bills 12 months (full policy term) of premium reports, rather than 11. If the billing system bills only 11 premium reports on the expiring period, the billing system applies the collateral deposit to the final audit. Next, it collects separate money for the renewal collateral.

Two use cases exist:

- **Deposit held for a single period** – Separate deposits are required for each period, with the deposit netted against the final amount due at end of period, with any excess returned.
- **Deposit held for a policy across periods** – Deposit is adjusted as premiums increase or decrease from period to period. The deposit is not returned at the end of a period unless the policy is canceled or not renewed.

The base configuration BillingCenter integration handles only the first case, a deposit held for a single period. All collateral held on the account displays as segregated separately for each policy period.

Deposits in policy change and reinstatement jobs

Policy-change jobs and reinstatement jobs recalculate the default deposit requirement based on the change in premium. In the PolicyCenter user interface, you can override the deposit percentage. When the job binds, PolicyCenter resends the full deposit amount to BillingCenter. This value is the total deposit amount required for the policy period, not the change in the deposit value.

When BillingCenter receives the deposit amount, BillingCenter updates the existing collateral requirement. There is only one active cash collateral item for each policy period at any one time. If the insured paid money toward the collateral requirement, the payment transfers to the current requirement.

Deposits in premium reports

PolicyCenter does not recalculate or send collateral information when premium report transactions complete. The deposit value does not change during these transactions.

Deposits in cancellation

Handling of deposits differs depending on when a cancellation occurs.

Midterm cancellation

In PolicyCenter, whenever the midterm cancellation binds, the policy period has a pending (scheduled or in progress) final audit. PolicyCenter resends the existing collateral requirement to BillingCenter. PolicyCenter does not recalculate the deposit amount. Even though the cost of the policy reduces after cancellation, PolicyCenter does not reduce the deposit requirement explicitly. After BillingCenter processes the final audit, the billing system can return any extra money to the insured.

Flat cancellation

For PolicyCenter, when the flat cancellation binds, the policy period does not have a pending final audit. In the process of BillingCenter processing the cancellation request, the collateral billing instruction releases the deposit by sending \$0 (zero dollars) as the new collateral requirement.

In BillingCenter, the active collateral requirement status changes to `closed`. BillingCenter releases money paid and held as “cash held by requirements” balance, which sets the money to the category of collateral cash held. The collateral cash is available for transfer. Typically, the insurer eventually transfers this amount to either pay off the final audit balance or apply the funds to the segregated collateral requirement of the renewal policy period.

Deposits in final audit

In PolicyCenter, whenever a final audit completes in PolicyCenter, the collateral billing instruction releases the deposit by sending a requirement of \$0 (zero dollars). This instruction applies to any final audit, both expiration and cancellation.

In BillingCenter, the active collateral requirement changes to `closed`. BillingCenter releases money paid and held as a “cash held by requirements” balance to the category of collateral cash held. The collateral cash is available for transfer. Typically, the insurer eventually transfers this amount to either pay off the final audit balance or apply the funds to the segregated collateral requirement of the renewal policy period.

An exception to the preceding rules occurs for PolicyCenter transactions after a final audit completes. PolicyCenter may produce transactions such as policy changes after a final audit completes. Such transactions result in reversing the final audit, followed by a policy change, followed by scheduling a new final audit. In such a case, an insurer does not want to bill the insured for a new deposit. After the final audit releases the deposit, the deposit remains released, unless a user reinstates a cancelled policy.

The base configuration PolicyCenter logic tracks the last calculated deposit amount and whether that deposit was already released. After releasing the deposit, PolicyCenter sends a deposit of \$0 (zero dollars) with each type of message from PolicyCenter to BillingCenter. After a deposit is released, only a reinstatement unreleases the deposit.

Billing implications of up-front payments

An *up-front payment* is a payment for a job that occurs before binding or issuing takes place. You must ensure that the billing system receives information about the payment at the time that the payment occurs. You must provide sufficient information about the payment for the billing system to identify the policy period to which the payment applies. If BillingCenter is your billing system, use the payments web service API provided by `PaymentAPI` to manage suspense payments. Create a suspense payment for a successful payment transaction. Remove a suspense payment for a successful credit transaction. Use the job number as the `OfferNumber` property of the `PaymentReceiptRecord` data transfer object that you send to BillingCenter.

Your billing system must be able to issue invoices that take account of receipt of any up-front payments for a policy period. You must take account of any delay that might occur in receiving the up-front payment before issuing invoices or starting delinquency.

See also

- “Payment integration” on page 547
- “Payments Web Service APIs (`PaymentAPI`)” in *BillingCenter Integration Guide*

Implementing the billing system plugin

If you want to use a billing system other than BillingCenter, create your own class that implements the billing system plugin interface: `IBillingSystemPlugin`. The billing system plugin is the main interface between PolicyCenter and its billing system.

Note: If you use BillingCenter, the base configuration plugin implementation handles these tasks. You do not perform the tasks in this topic.

Your plugin implementation must implement all the methods defined by the plugin interface. This plugin handles a wide variety of billing-related tasks.

The most important thing to know about implementing the billing system plugin is that PolicyCenter billing requests typically happen asynchronously. The typical flow of a billing request is as follows:

1. A user changes something in PolicyCenter. This change causes a messaging event to fire. Possible messaging events are:
 - A custom messaging event that some code triggers, for example, completing a policy change.
 - Automatic messaging events that trigger because of changes in Guidewire entity instances, for example, `PolicyPeriod` or `Contact`.
2. As a consequence, PolicyCenter rule sets trigger.
3. The Event Fired rules detect what changed and creates event messages representing what type of change happened and submits them to the messaging event queue.
4. As the bundle for the original change commits, the messages commit to the database.
5. In a separate task on the batch server only, messaging plugins send messages. In the case of billing, message transports specific to billing attempt to send each message. The way it sends a message is to call methods on the currently enabled `IBillingSystemPlugin` plugin implementation.
6. Your billing system plugin method must perform its task synchronously with respect to the caller of the plugin method. However, the call typically happens inside messaging transport code, which runs asynchronously with respect to the original change in the PolicyCenter user interface and the application database.

IMPORTANT Your billing system plugin code is called as part of an asynchronous task that the messaging system manages. It is not called as part of the original database transaction for the change that triggers the call to the billing system.

7. Your plugin code must always act on the `PolicyPeriod` passed to it. Because this is asynchronous, theoretically multiple changes might happen to a policy before all the messages are processed by the messaging system. By the time your plugin is called, it might actually not be the most recently bound version of the policy. However, you must always handle the requests in the order you receive them. Just use the `PolicyPeriod` reference that is the method parameter, and it is possible you might get additional updates or changes in future calls to your plugin.

See also

- “Billing integration overview” on page 501
- “Asynchronous communication between PolicyCenter and billing system” on page 504
- *Installation Guide*

Account management

The billing system plugin has several methods related to accounts.

Your `createAccount` method tells your billing system to create an account associated with the policy period for a particular transaction. As a method parameter, this plugin method gets the PolicyCenter `Account` entity instance that represents the new account. The `Account` entity properties contain the data for the new billing system account.

Note: PolicyCenter always calls this method asynchronously with respect to the user interface. The messaging code calls out to the plugin.

Use the account number of an account, not its public ID, to uniquely identify the account in both systems. The method contains a second parameter, which is the billing system transaction ID.

The transaction ID is an identifier that PolicyCenter creates. Your plugin must track this billing system transaction ID. If PolicyCenter requests the same transaction ID twice, the billing system or your plugin that represents it must detect that PolicyCenter requested it twice. If you receive the same transaction ID more than once, you must ignore the duplicate requests.

Your implementation of the `updateAccount` method likely is similar to your `createAccount` method. It takes the same method parameters, an `Account` entity instance and a transaction public ID `String`. However, instead of creating a new account, PolicyCenter calls this method to notify the billing system that some account information changed. Use the current properties on the `Account` entity instance to update the billing system version of the account. Use the account's public ID (the `PublicID` property) to uniquely identify the account in both systems.

Your `accountExists` method asks your billing system whether an account exists, based on the account number. Return `true` if it exists, otherwise return `false`. PolicyCenter does not track whether the billing system already knows about an account, so it always asks and then creates it if necessary.

Policy period management

The billing system plugin provides several methods related to policy periods.

New policy periods

Your plugin's `createPolicyPeriod` method tells your billing system to create a policy period associated with a particular transaction. As a method parameter, this plugin method gets the PolicyCenter `PolicyPeriod` entity instance that represents data in the new policy period. However, the billing system view of a policy period typically is significantly simpler than the PolicyCenter view of the same policy period.

Note: PolicyCenter always calls this method asynchronously with respect to the user interface. The messaging code calls out to the plugin.

Use the policy period's policy number and term number to uniquely identify the policy period in both systems. The method contains a second parameter, which is the billing system transaction ID.

The transaction ID is an identifier that PolicyCenter creates. Your plugin must track this billing system transaction ID. If PolicyCenter requests the same transaction ID twice, the billing system or your plugin that represents it must detect that PolicyCenter requested it twice. If you receive the same transaction ID more than once, you must ignore the duplicate requests.

If you have any data model extension properties on the PolicyCenter version of `PolicyPeriod`, consider whether any are appropriate for your billing system.

PolicyCenter calls this method as part of binding a new policy:

- If you perform a bind and issue as one user interface action, PolicyCenter calls `createPolicyPeriod` only.
- If you perform just a bind in the user interface, PolicyCenter creates the new `PolicyPeriod` locally and then calls the billing system plugin `createPolicyPeriod` method. For an issuance request that occurs later, PolicyCenter calls the plugin's `issuePolicyChange` method.

See also

- “Billing system notifications of PolicyCenter policy actions” on page 536

Cancellation and reinstatement

PolicyCenter starts some types of cancellations:

- Cancellation as part of a rewrite request.
- Cancellation at the insured's request
- Cancellation due to the insured lied in their application or violated the contract
- Cancellation because the insured chooses not to take a renewal and the renewal might already be bound

To the billing integration code, all these cases are the same.

Your `IBillingSystemPlugin` plugin must handle these types of cancellations.

Note: PolicyCenter always calls this method asynchronously with respect to the user interface. The messaging code calls out to the plugin.

The billing system starts other types of cancellations. For example, cancellation for non-payment. If the billing system detects overdue invoices, the billing system starts a delinquency process and eventually tells PolicyCenter to

cancel the policy. However, the billing information flow is independent of which system starts the cancellation. In other words, whenever the cancellation completes, PolicyCenter sends the billing implications to BillingCenter.

Note: BillingCenter starts cancellation by calling PolicyCenter web services.

If you implement the `IBillingSystemPlugin` plugin, your plugin's `cancelPolicyPeriod` method notifies your billing system of a policy cancellation. This method's parameters are a policy period as a `PolicyPeriod` entity instance and a billing system transaction ID.

The transaction ID is an identifier that PolicyCenter creates. Your plugin must track this billing system transaction ID. If PolicyCenter requests the same transaction ID twice, the billing system or your plugin that represents it must detect that PolicyCenter requested it twice. If you receive the same transaction ID more than once, you must ignore the duplicate requests.

Cancellation can often be a complex process with legally required notification delays. PolicyCenter does not call `cancelPolicyPeriod` at the time of a request to initiate cancellation. PolicyCenter calls this method after binding the cancellation transaction. Typically, this plugin method sends a cancellation billing instruction or equivalent command to your billing system in whatever way is appropriate. This method returns no value.

To support reinstatement, your plugin's `issueReinstatement` method notifies your billing system of a policy reinstatement. It takes the same arguments as `cancelPolicyPeriod` and returns no value.

See also

- “Policy cancellation and reinstatement web services” on page 116

Rewrite

Your plugin must also prepare for a policy rewrite request and send that information to your billing system.

PolicyCenter calls the `rewritePolicyPeriod` method to rewrite the policy period and `rewritePolicyPeriod` must send that information to your billing system. Its two method parameters are a policy period (`PolicyPeriod`) and a billing system transaction ID.

Note: PolicyCenter always calls this method asynchronously with respect to the user interface. The messaging code calls out to the plugin.

The transaction ID is an identifier that PolicyCenter creates. Your plugin must track this billing system transaction ID. If PolicyCenter requests the same transaction ID twice, the billing system or your plugin that represents it must detect that PolicyCenter requested it twice. If you receive the same transaction ID more than once, you must ignore the duplicate requests.

Getting period information from the billing system

Your plugin must be able to retrieve billing information to deliver to PolicyCenter whenever PolicyCenter needs the information. If PolicyCenter needs billing information, it calls this plugin's `getPeriodInfo` method. This method is different from most other methods because PolicyCenter always calls this method synchronously with respect to the user interface.

Your plugin must retrieve this information from the billing system and encapsulate this information in a `PolicyPeriodBillingInfo` object. The fully qualified class name of this class is `gw.api.billing.PolicyPeriodBillingInfo`.

PolicyCenter uses this information as part of performing renewal. PolicyCenter tries to preserve the billing information for the period for the renewal period.

Billing system notifications of PolicyCenter policy actions

User actions in PolicyCenter can have implications for an external billing system. For example, a policy change might change the premium for a billing account. PolicyCenter calls various methods for each action, and your plugin must notify the billing system to take the appropriate action.

Your plugin might populate properties in an object that represents the policy period and recent changes, and then send that request to your billing system in a web service request. The base configuration billing system plugin does these tasks to notify BillingCenter.

If you do not use BillingCenter, you must handle all of the events for which PolicyCenter calls `IBillingSystemPlugin` methods.

All methods take the same parameters: a policy period (`PolicyPeriod`) and a billing system transaction ID.

The transaction ID is an identifier that PolicyCenter creates. Your plugin must track this billing system transaction ID. If PolicyCenter requests the same transaction ID twice, the billing system or your plugin that represents it must detect that PolicyCenter requested it twice. If you receive the same transaction ID more than once, you must ignore the duplicate requests.

Billing system plugin method requirements

The following table lists properties that you might want to send to your billing system and other important information about `IBillingSystemPlugin` methods.

Action	Method name	Description
All actions	All methods	For each action, you typically need to send the following basic policy period data: <ul style="list-style-type: none">• <code>period.PeriodId.Value</code>• <code>period.PolicyNumber</code>• <code>period.EditEffectiveDate.toCalendar()</code>• <code>ChargeInfoUtil.getChargeInfos(period)</code>
Policy change	<code>issuePolicyChange</code>	Issue a policy change in the billing system. PolicyCenter does not notify the billing system about all changes to the policy. PolicyCenter does notify the billing system about the following policy changes: <ul style="list-style-type: none">• Premium changes• Base state changes• Primary insured changes In addition to basic policy period data, you typically need to send the following policy period data: <ul style="list-style-type: none">• <code>period.PolicyChange.Description</code>• <code>period.BaseState.Code</code>• <code>period.PeriodStart.toCalendar()</code>• <code>period.PeriodEnd.toCalendar()</code>• <code>period.PrimaryNamedInsured.AccountContactRole.AccountContact .Contact</code> (a <code>Contact</code> entity)

Action	Method name	Description
Issuance	issuePolicyChange	<p>PolicyCenter calls the <code>issuePolicyChange</code> method for Issuance as well as policy changes. PolicyCenter does not call this method as part of the bind process, but calls it for issuance that happens as a separate action. If the user performs just a bind, PolicyCenter creates the new <code>PolicyPeriod</code> locally and then calls the billing system <code>createPolicyPeriod</code> method. If the user performs a bind and issue as one user interface action, PolicyCenter calls <code>createPolicyPeriod</code> only. If the user performs bind only, PolicyCenter calls <code>createPolicyPeriod</code> only. Any issuance request that occurs later results in PolicyCenter calling <code>issuePolicyChange</code>. See the policy change row for properties you might want to send to your billing system.</p>
Premium report	issuePremiumReport	<p>Issue a Premium Report in the billing system</p> <p>In addition to basic policy period data (see note before this table), you will probably need to send the following policy period data:</p> <ul style="list-style-type: none"> • <code>period.Audit.AuditInformation</code> • <code>period.Audit.AuditInformation.AuditPeriodEndDate.toCalendar()</code> • <code>period.Audit.AuditInformation.AuditPeriodStartDate.toCalendar()</code> • <code>period.Audit.PaymentReceived</code>
Renew	renewPolicyPeriod	<p>Renew a policy period in Billing System. This plugin only notifies the billing system. This plugin does not change the fundamental logic within PolicyCenter for how a policy renews.</p> <p>In addition to basic policy period data (see note before this table), you will probably need to send the following policy period data:</p> <p>See the Policy Change row for properties you might want to send to your billing system.</p>
Term confirmation	updatePolicyPeriodTermConfirmed	Confirms a policy period term. This method differs from the others in this table in that it does not notify the billing system. Instead, the billing system notifies PolicyCenter that a specified policy period term has been confirmed by calling the <code>PolicyRenewalAPI</code> web service method <code>confirmTerm</code> . The web service method calls this plugin method

Action	Method name	Description
		which sets the policy period term's <code>TermConfirmed</code> field to true.
Final audit	<code>issueFinalAudit</code>	Issue a Final Audit in Billing System. You might need only the basic policy period data to send to the billing system. See the note before this table for the property list.
Schedule final audit	<code>scheduleFinalAudit</code>	Schedule a final audit on a policy period in your billing system. In addition to basic policy period data (see note before this table), you will probably need to send the following policy period data: <ul style="list-style-type: none"> • <code>period.PolicyNumber</code> • <code>period.PeriodId.Value</code>
Waive final audit	<code>waiveFinalAudit</code>	Schedule a final audit on a policy period in your billing system. You might need only the basic policy period data to send to the billing system. See the note before this table for the property list. In addition to basic policy period data (see note before this table), you will probably need to send the following policy period data: <ul style="list-style-type: none"> • <code>period.PolicyNumber</code> • <code>period.PeriodId.Value</code>

See also

- “New policy periods” on page 535

Getting the billing level

The `getBillingLevel` method in the billing system plugin gets the billing level for the specified account or returns a default billing level if the account does not exist on the billing system.

The billing level specifies how to bill policies on the account. In the base configuration, billing levels are defined in the `BillingLevel.tti` typelist. The values are:

- **Account** – Group policies together into a single account bill. Any positive balances in designated `Unapplied` T-accounts are transferred to the default `Unapplied` account. Appears as **Account** in **Billing Level** on the **Payment** screen.
- **PolicyDefaultUnapplied** – Bill each policy individually without cash separation. Appears as **Policy (Separate Funds by Account)** in **Billing Level** on the **Payment** screen.
- **PolicyDesignatedUnapplied** – Bill each policy individually with cash separation. Appear as **Policy (Separate Funds by Policy)** in **Billing Level** on the **Payment** screen.

The `getBillingLevel` method retrieves the billing level on the account from the billing system. This method can set a default level if the billing level has not been set for the account. This occurs when creating a policy on an account that does not exist in the billing system.

The `StandAloneBillingSystemPlugin` plugin implementation always bills each policy individually; `getBillingLevel` returns `PolicyDesignatedUnapplied`.

The `BCBillingSystemPlugin` plugin implementation retrieves the billing level from `BillingCenter`. If the policy is on an account that does not exist in the billing system, the default is to bill each policy individually,

`PolicyDesignatedUnapplied`. The default value is specified in the `getDefaultValue` method. If you wish to change the way the default value is specified, you can modify this method.

Producer management

PolicyCenter tracks various types of organizations with the `Organization` entity. Every organization has an organization type in its `Type` property, which holds a `BusinessType` typecode. If the typecode of the organization type has a `Category` property of `BusinessTypeCategory.producer`, PolicyCenter considers this organization an external producer. An external producer organization has a corresponding entity instance in the billing system. PolicyCenter sends all new or changed producer organization records to the billing system. This includes PolicyCenter organizations such as Agency, Broker, or Managing General Agent.

Your billing system might use different terminology for this category of organization. For the purposes of the billing system integration, PolicyCenter and its documentation considers these organizations producers.

Producers

To accommodate new producers, implement the `createProducer` method, which takes as arguments the `Organization` entity instance and the billing system transaction ID.

The transaction ID is an identifier that PolicyCenter creates. Your plugin must track this billing system transaction ID. If PolicyCenter requests the same transaction ID twice, the billing system or your plugin that represents it must detect that PolicyCenter requested it twice. If you receive the same transaction ID more than once, you must ignore the duplicate requests.

Your method must return the billing system version of the public ID for this `Organization`. Your method implementation must notify the external system of the new producer in whatever way is appropriate.

Your code would typically use the following `Organization` entity properties:

- `Name`
- `AgencyBillPlanID`
- `PublicID`
- `Tier.Code`
- `Contact` (copy all the properties on the `Contact` entity instance for this producer)

Your code might use additional properties on `Organization`.

Update producers

You also must prepare for changes to the producer within PolicyCenter. If it changes, PolicyCenter calls the billing system plugin to send updates to the billing system. Implement the `updateProducer` similar to your `createProducer` method. It takes the same parameters but return nothing. Use the PolicyCenter public ID to identify the producer in the external system.

You also must implement the method `producerExists`, which takes a producer public ID and returns `true` or `false` to indicate if the producer exists in the billing system.

Syncing producers

Your plugin must implement the `syncOrganization` method to synchronize an `Organization` object with the latest values from the corresponding producer in the billing system. The method takes a `Organization` object and returns nothing.

Producer codes

Your plugin must implement a method to notify the billing system of a new producer code. Implement the `createProducerCode` method. Its arguments are a producer code (`ProducerCode`) and a billing system transaction ID.

The transaction ID is an identifier that PolicyCenter creates. Your plugin must track this billing system transaction ID. If PolicyCenter requests the same transaction ID twice, the billing system or your plugin that represents it must

detect that PolicyCenter requested it twice. If you receive the same transaction ID more than once, you must ignore the duplicate requests.

Your method must return the billing system version of the public ID for this producer code.

Your code would typically send the following `ProducerCode` entity properties:

- `PublicID`
- `Code`
- `ProducerStatus`
- `producerCode.Organization.PublicID`
- `producerCode.CommissionPlanID`

Updating producer codes

Your plugin must implement the `updateProducerCode` method, which takes the same arguments as the `createProducerCode` method, but does not return a value.

Synchronizing producer codes

Your plugin must implement the `syncProducerCode` method to synchronize a `ProducerCode` object with the latest values from the billing system. The method takes a `ProducerCode` object and returns nothing.

Getting billing methods available for a producer code

Your plugin must implement the `getAvailableBillingMethods` method, which takes a producer code public ID and returns an array of `BillingMethod` entities that the producer code supports.

Agency bill plan availability retrieval

Your billing plugin must be able to retrieve all the possible agency billing plans from your billing system.

Implement the plugin method `retrieveAllAgencyBillPlans`, which takes no arguments. It returns an array of `AgencyBillPlanSummary` objects. The fully qualified class name is `gw.api.billing.AgencyBillPlanSummary`.

This method is notably different from most other methods because PolicyCenter always calls this method synchronously with respect to the user interface. PolicyCenter uses this method to populate a drop down list for billing plans if using agency bill.

PolicyCenter allows the user to set this billing plan information initially only. For any later changes, you must directly change the information in the billing system.

Set the following properties on each `AgencyBillPlanSummary` object

- `Name` – The name of the plan
- `Id` – The ID for the plan

Commission plan management

Your billing plugin must be able to retrieve all the commission plans from your billing system. Implement the plugin method `retrieveAllCommissionPlans`, which takes no arguments. It returns an array of `CommissionPlanSummary` objects. The fully qualified class name is `gw.api.billing.CommissionPlanSummary`.

This method is notably different from most other methods because PolicyCenter always calls this method synchronously with respect to the user interface. PolicyCenter uses this method to populate a drop down list for commission plans if using agency bill.

PolicyCenter allows the user to set this commission plan information initially only. For any later changes, you must directly change the information in the billing system.

Set the following properties on each `CommissionPlanSummary` object

- `Name` – The name of the plan
- `Id` – The ID for the plan
- `AllowedTiers` – An array of `Tier` typecodes that this commission plan supports

Payment plans and installment previews

Prior to binding and after quoting, the **Payment** screen displays the available payment plans and supports opening a preview screen for the installments for a selected plan. Your billing plugin must be able to retrieve both these types of information. PolicyCenter uses a `PaymentPlanSummary` entity instance to store summary information about the payment plan for a bound submission. Your billing plugin must be able to create this entity instance, populating its properties from an instance of a Gosu class that has a similar structure.

Retrieving payment plans from a billing system

Your billing plugin must be able to retrieve all the allowed payment plans from your billing system for a quoted `PolicyPeriod`. PolicyCenter uses this information in the user interface on the **Payment** screen. PolicyCenter displays the list of payment plans so the user can choose one of the payment plans for PolicyCenter jobs that create a new period. Job types that create a new period are submission, renewal, and rewrite.

Retrieving payment plan data objects

Implement both signatures of the following plugin method:

- `retrieveAllPaymentPlans` – Retrieves payment plans for a policy period or for a particular billing method that an alternate billing account uses

Both methods return an array of Gosu objects that implement the `PaymentPlanData` interface. One method takes a `PolicyPeriod` entity instance as a parameter. The other method takes a `BillingMethod` entity instance, an account number `String` for the alternate billing account, and the `Currency` to use for the payment plan as parameters. Take these parameter values from a particular `PolicyPeriod` entity instance.

Populating payment plan data objects

The `gw.plugin.billing` package contains the `PaymentPlanData` base interface and two further interfaces, `InstallmentPlanData` and `ReportingPlanData`, that extend `PaymentPlanData` for installment and reporting plan types. An abstract class, `AbstractPaymentPlanData`, implements the `PaymentPlanData` base interface. Two implementations that integrate with BillingCenter exist in the same package. Use these implementations to guide the development of integration to your own billing system:

- `InstallmentPlanDataImpl` – Use for installment plans
- `ReportingPlanDataImpl` – Use for report payment plans

The fully qualified class names look like `gw.api.billing.PaymentPlanClass`.

To support displaying the available payment plans in the user interface, set the following properties on each `PaymentPlanData` object:

- `BillingId` – The ID that the billing system uses for the plan.
- `PaymentPlanType` – The `PaymentMethod` type code of the plan, either `Installments` or `ReportingPlan`.
- `Notes` – Optional. A `String` value containing notes related to the plan.

For an installment payment plan, set the following additional properties:

- `Name` – The name of the plan.
- `DownPayment` – A non-negative `MonetaryAmount` value that is the down payment of the plan, typically the largest installment.
- `Fee` – A non-negative `MonetaryAmount` value that is the fee to pay at the same time as each installment of the plan.
- `InvoiceFrequency` – The `BillingPeriodicity` type code of the plan, for example monthly or quarterly.

- **Installment** – A non-negative `MonetaryAmount` value that is the installment payment of the plan.
- **OneTimeFee** – A non-negative `MonetaryAmount` value that is typically the fee to pay at the same time as the down payment.
- **Total** – A non-negative `MonetaryAmount` value that is the total payment of the plan. This value is the sum of the down payment, all fees, all taxes, and all installments combined.
- **TotalFees** – A non-negative `MonetaryAmount` value that is the total fees to pay for the plan. This value is the sum of the one-time fee and all the installment fees.

For a reporting payment plan, set the following additional properties:

- **Name** – The name of the plan if your class does not extend `ReportingPlanDataImpl`. `ReportingPlanDataImpl` sets the `Name` property to the `Name` property of the audit schedule of the reporting plan.
- **ReportingPatternCode** – The code that the billing system uses for the audit schedule of the reporting plan.

Saving the selected payment plan for the policy period

To save the selected payment plan when the user binds the `PolicyPeriod`, set the `SelectedPaymentPlan` property of the `PolicyPeriod` by calling the `PaymentPlanData` method `createPaymentPlanSummary`. Pass a reference to the current bundle to this method. For example, use code similar to the following line:

```
newPolicyPeriod.SelectedPaymentPlan = selectedPlan.createPaymentPlanSummary(currentBundle)
```

Use the properties on the selected `PaymentPlanData` object to set the values of the properties on the `PaymentPlanSummary` entity instance. The `PaymentPlanSummary` entity instance does not have separate implementations for installment and reporting plan types as `PaymentPlanData` does. Instead, the data model requires non-null values only for the properties that both types use. The properties that only installment or reporting plan types use are optional and can have null values. Your implementation must set these properties appropriately.

In the base configuration, the implementation of the `createPaymentPlanSummary` method sets the properties that both installment and reporting plan types of `PaymentPlanSummary` entity use. This method then calls the `doCreatePaymentPlanSummary` method. The classes that extend `AbstractPaymentPlanData` set the additional properties that the payment plan type uses.

See also

- *Gosu Reference Guide*

Accessing the selected payment plan

To retrieve the selected payment plan for a bound `PolicyPeriod`, create a `PaymentPlanData` object of the appropriate type and call the `doPopulateFromPaymentPlanSummary` method. If the billing system changes the properties of the payment plan, PolicyCenter does not reflect these changes until you start a new transaction on the policy. The payment plans available for the new policy period use the current plan information from the billing system.

Retrieving installment previews from a billing system

Similar to the requirements for payment plans, your billing plugin must be able to retrieve installment previews from your billing system. Installment previews are an accurate calculation of billing installments for a given policy period with a given payment plan. PolicyCenter uses this information when a user clicks the **Schedule** button in the **Schedule** column for a payment plan to show the user what installments to expect.

Implement the following plugin method:

- `retrieveInstallmentsPlanPreview` – Retrieves previews of installments for a new policy, policy change, policy renewal, or policy rewrite job. This method takes a `PolicyPeriod` entity instance and returns an array of `PaymentPreviewItem` objects.

PolicyCenter always calls this method synchronously with respect to the user interface. This method takes a `PolicyPeriod` entity instance and returns an array of `PaymentPreviewItem` objects. The fully qualified class name of this class is `gw.api.billing.PaymentPreviewItem`.

Your implementation of this method must set the following properties on each `PaymentPreviewItem` object:

- `DueDate` – The due date
- `Type` – A `String` that represents the type of payment
- `Amount` – The amount of money, as a `MonetaryAmount` value

The methods described in this topic get installment previews, not payment plans on a policy period. To get payment plans, you must retrieve the payment plans from the billing system.

See also

- “Retrieving payment plans from a billing system” on page 542

Updating contacts

In the base configuration, PolicyCenter is designed to use a contact management system as the system of record for contact management. If a contact associated with billing information changes in PolicyCenter, PolicyCenter determines whether to notify a contact management system by calling the method `gw.contact.ContactEnhancement.ShouldSendToContactSystem`. If the method returns `true`, the contact updates are sent to the external contact system.

If you do not use an external contact management system, comment out the contents of the `shouldSendContactUpdate` method and change its return value to `false`. For example:

```
property get ShouldSendToContactSystem() : boolean {
    // return this.AutoSync == AutoSync.TC_ALLOW
    // and not this.ID.Temporary
    // and (isOnAccountWithLinkContacts() or isReinsuranceParticipant())
    return false;
}
```

This feature applies to the following contacts:

- New or changed contacts for a producer
- New or changed contact for an account holder
- New or changed contacts for a policy period billing contact
- New or changed contacts for an policy period primary insured.

Changes to non-primary insured contact information does not trigger this request.

Note: PolicyCenter always calls this method asynchronously with respect to the user interface. The messaging code calls out to the plugin.

If you use a contact management system, your implementation of the `BCBillingSystemPlugin.updateContact` method must update the contact information for a `Contact` entity instance. The parameters to the method are a `Contact` entity instance and the billing system transaction ID.

The transaction ID is an identifier that PolicyCenter creates. Your plugin must track this billing system transaction ID. If PolicyCenter requests the same transaction ID twice, the billing system or your plugin that represents it must detect that PolicyCenter requested it twice. If you receive the same transaction ID more than once, you must ignore the duplicate requests.

Use the public ID of the contact to uniquely identify the contact in the external billing system.

Note: If you remove a role from a contact in the base configuration implementation, PolicyCenter does not notify the billing system that the contact no longer has that role.

Other billing system plugin methods

There are other methods on the billing plugin interface:

- `addPaymentInstrumentTo` – Adds a payment instrument to an account and returns a billing system instrument. For example, an insured's credit card number or a token that represents it in an external payment system.
- `getExistingPaymentInstruments` – Returns an array of billing system instruments.
- `getInvoiceStreams` – Retrieves all invoice streams for a given account.
- `getSubAccounts` – Retrieves all subaccounts of the account with the given account number. If the account does not exist, returns an empty array. The search for subaccounts is recursive and returns subaccounts of subaccounts also.
- `searchForAccounts` – Search for accounts in the billing system by using search criteria.

Implementing the billing summary plugin

To support the PolicyCenter billing summary screens (one in policy file, one in account file), implement the `IBillingSummaryPlugin` plugin interface. PolicyCenter uses the information that this plugin returns only as part of the billing summary screen. PolicyCenter does not use the information as part of binding, issuing, changing, or renewing a policy. You can remove the billing summary screen from the user interface and PolicyCenter would continue to work normally.

Note: If you use BillingCenter, the base configuration plugin implementation handles these tasks. You do not perform the tasks in this topic.

See also

- “Billing integration overview” on page 501
- *Installation Guide*

Interfaces used by the billing summary plugin

The methods in the billing summary plugin interface must return instances of classes that implement one of several pre-defined interfaces. You define your own Gosu classes that are concrete implementations of these Java interfaces, which are in the `gw.plugin.billing` package. To write a new implementation of the billing summary plugin itself (`IBillingSummaryPlugin`), write classes that are concrete implementations of all of the following interfaces:

- `BillingAccountInfo`
- `BillingInvoiceInfo`
- `BillingPeriodInfo`

These interfaces define methods. Many of the methods are property getter methods. In Gosu, these getters are exposed as properties. For example, the `BillingPeriodInfo` interface (defined in Java) has a `getPolicyTermInfos` method. Write a new Gosu class that implements this interface, which would implement a getter for the `PolicyTermInfos` property. Gosu code can access its `PolicyTermInfos` property directly or by calling the `getPolicyTermInfos` method. You can also add additional getters and setters for other properties that you create.

Other methods on these interfaces are regular methods and not getter methods. For example, `BillingPeriodInfo` interface has a `findPolicyPeriod` method.

Note: Studio can provide the full list of methods, property getters, and property setters to implement on each interface. To implement an interface, first define a class that extends the interface. The Gosu editor offers a quick fix to create method declarations for all the interface methods and property getters and setters.

PolicyCenter includes two implementations of `IBillingSummaryPlugin`:

- `BCBillingSummaryPlugin` – Base configuration plugin implementation for PolicyCenter integration with BillingCenter.
- `StandAloneBillingSummaryPlugin` – Example plugin implementation that mimics integration with a third-party billing system.

In Studio, view these classes as examples of implementing the `IBillingSummaryPlugin` plugin interface and its additional interfaces `BillingAccountInfo`, `BillingInvoiceInfo`, and `BillingPeriodInfo`.

Getting policies billed to accounts

To retrieve all open policies billed to an account, implement the `getPoliciesBilledToAccount` method in the billing summary plugin. This method takes an account number `String` and returns an array of open policy periods that bill to the account. Each array member is an instance of a class that implements the `BillingPeriodInfo` interface.

```
BillingPeriodInfo[] getPoliciesBilledToAccount(String accountNumber);
```

Retrieving account billing summaries

To retrieve a billing summary for an account, implement the `retrieveAccountBillingSummary` method. This method takes an account number `String` and returns an instance of a class that implements the `BillingAccountInfo` interface. The class instance contains billing information for the account.

```
BillingAccountInfo retrieveAccountBillingSummary(String accountNumber);
```

Retrieving account invoices

To retrieve the account invoices associated with an account, implement the `retrieveAccountInvoices` method. This method takes an account number `String` and returns an array of instances of a class that implements the `BillingInvoiceInfo` interface. Each instance contains information about the invoice for the account.

```
BillingInvoiceInfo[] retrieveAccountInvoices(String accountNumber);
```

Retrieving billing summaries for policy periods

To retrieve billing summaries for all open policy periods associated with an account, implement the `retrieveBillingPolicies` method. This method takes an account number `String` and returns an array of instances of a class that implements the `BillingPeriodInfo` interface. Each array member contains a billing summary for a policy period associated with the account.

```
BillingPeriodInfo[] retrieveBillingPolicies(String accountNumber);
```

Retrieving policy billing summary

To retrieve a billing summary for a policy period, implement the `retrievePolicyBillingSummary` method. This method takes a policy number `String` and the term of the policy period and returns an instance of a class that implements the `BillingPeriodInfo` interface. This instance contains the billing summary for the policy period.

```
BillingPeriodInfo retrievePolicyBillingSummary(String policyNumber, int termNumber);
```

Payment integration

Payment integration involves setting up PolicyCenter communication with a payment system. The types of communication include:

- Adding and using payment instruments that a payment system uses for billing
- Using a payment gateway to make an up-front payment

Overview of integration with a payment system

From the **Payment** screen of some jobs, a user can collect information about the payment source for immediate or future payments. Typically, PolicyCenter transfers control to an external third-party system to collect payment source information. You configure PolicyCenter to use an external third-party system to collect and store payment source information to avoid storing that information in the PolicyCenter database.

For example, the insured can specify in the external payment processing system to use a specific credit card number for immediate or future billing. The payment system collects the insured's private financial information for the transaction and returns to PolicyCenter a token that represents the payment instrument. Whenever PolicyCenter sends this token to the billing system, the billing system uses that token to initiate a payment in the same third-party payment system. Alternatively, PolicyCenter can communicate with an external payment gateway to create the token without taking a payment.

PolicyCenter can receive a deposit for a reporting plan or a down payment for an installment plan as an up-front payment. Similarly, PolicyCenter can take an additional up-front payment as changes, renewals, and rewrites modify the amount of the deposit or down payment that the policy requires. These payments can use the same payment instrument as other payments for the job, such as installments, or a different instrument. For electronic payments, PolicyCenter transfers control to an external payment gateway that processes the payment. The gateway returns information about the success or failure of the payment to PolicyCenter. PolicyCenter sends the information about successful up-front payments to the billing system.

Entities and classes for tracking payments

PolicyCenter uses the following Gosu objects and data model entities to track the usage of payment instruments and payments for a policy period:

Item	Type	Usage
PolicyPeriodBillingInstructionsManager	Gosu object	Tracks the payments that the selected payment plan requires and the up-front payments that the user enters.
BillingPaymentInstrumentImpl	Gosu object	Provides basic information about each payment instrument for a policy. The billing system is the system of record for these payment instruments. The PolicyPeriodBillingInstructionsManager object uses these objects in communication with both an external payment system and an external payment gateway.
TokenizedCreditCardDetails	Gosu object	Provides two <code>String</code> values. One value, which typically has a format similar to <code>xxxx-1234</code> , identifies a credit card on the Payment user interface. The other value is a token that identifies the card to the payment gateway.
PaymentInformationBase	Gosu object	Transfers information about a payment to the payment gateway for processing.
PaymentGatewayTransaction	Data model entity	Associates an up-front payment request with a PolicyPeriod entity instance. PolicyCenter deletes the PaymentGatewayTransaction entity instance when it receives confirmation that the payment gateway has processed the payment.
PaymentGatewayResponseBase	Gosu object	Contains the information that a payment gateway returns from a request that PolicyCenter makes.
PaymentGatewayResponseProcessor	Gosu object	Processes the information in a PaymentGatewayResponseBase object for the PolicyPeriodBillingInstructionsManager object.

Item	Type	Usage
UpFrontPayment	Data model entity	Stores an up-front payment for a job. The <code>PaymentGatewayResponseProcessor</code> object creates this entity for an up-front payment on receiving a successful response from the payment gateway.

Integration points to external payment systems

PolicyCenter provides integration points to two external systems for setting up a payment instrument:

- External payment system – The PolicyCenter base configuration supports creating a payment instrument and sending information about the instrument to the billing system.

For demonstration purposes, the base configuration calls a demonstration version of a payment system that PolicyCenter itself implements with PCF pages. You can implement a similar system if you want to directly collect the payment source information and use it to initiate payments. In such an approach, you would not use tokens. You would directly contact the payment processor with the payment source information that you collected and stored.

- External payment gateway – The PolicyCenter base configuration supports receiving an up-front payment that uses a new or existing payment instrument and sending information about the instrument and payment to the billing system.

For demonstration purposes, the base configuration calls a demonstration version of a payment gateway system that PolicyCenter itself implements with PCF pages. You can implement a similar system if you want to directly collect the payment source information and use it to initiate payments. In such an approach, you would not use tokens. You would directly contact the payment processor with the payment source information that you collected and stored.

IMPORTANT External payment systems and payment gateways return only partial information for payment sources. For example, payment systems do not return entire credit card numbers. Instead, payment systems return tokens as identifiers that let PolicyCenter or its billing system initiate a payment without requiring the entire credit card number or other authentication information.

WARNING Check with your legal department about all regulatory requirements relating to credit card information before going to production.

See also

- “Implementing the billing system plugin” on page 533

Accessing an external payment system to add a new payment instrument

The following steps show the tasks that you perform and the actions that PolicyCenter takes to access an external payment system:

1. You start a submission, renewal, or rewrite job.
2. You quote the policy and navigate to the **Payment** screen before binding the policy. PolicyCenter displays billing methods, payment methods, and associated installment or reporting plans retrieved from the billing system.
3. You select a billing method and payment schedule, and then preview payment plans retrieved from the billing system. This part of the wizard is similar for all lines of business. In the Payment screen, the user interface that relates to payments is available if you select **Custom Billing Options** and the **New** option for **Invoicing**. The PCF file `BillingInvoiceStreamInputSet.pcf` provides the user interface for the payments. This PCF file lists payment-related information and provides an option to add payment information for recurring payments. If you do not take the down-payment amount as an up-front payment, PolicyCenter uses the same payment instrument for the down payment as for recurring payments.

4. PolicyCenter uses the registered billing system plugin (`IBillingSystemPlugin`) implementation to request the known payment instruments from the billing system. PolicyCenter calls its plugin method `getExistingPaymentInstruments`.
5. To add a new payment instrument, in `BillingInvoiceStreamInputSet`, you click the **Add** button. The action that the **Add** button takes is to call a method called `externalPaymentLocation` on the `gw.billing.PolicyPeriodBillingInstructionsManager` object that the page uses. The `externalPaymentLocation` method calls a PCF exit point to navigate to the payment system user interface. Typically, the payment system is on a separate physical computer across the Internet.
 - For demonstration purposes, the base configuration of this function calls a demonstration version of a payment system that PolicyCenter itself implements.
 - For a production environment, you must change the code for the `externalPaymentLocation` method to contact your real payment system.
6. The last step in `externalPaymentLocation` is to invoke the PCF exit point called `CreatePaymentInstrument`. Both the demonstration and a real payment system use this exit point. The following table compares the two payment systems:

Step	Demonstration payment system	Real payment system
1.	The <code>externalPaymentLocation</code> method sets the value of the <code>paymentSystemURL</code> property on <code>CreatePaymentInstrument</code> to the demonstration payment system.	The <code>externalPaymentLocation</code> method sets the value of the <code>paymentSystemURL</code> property on <code>CreatePaymentInstrument</code> to the real payment system.
2.	The <code>CreatePaymentInstrument</code> exit point transfers control to the <code>NewPaymentInstrument</code> entry point to display <code>CreateSamplePaymentInstrument.pcf</code> , which is the demonstration payment collection page. Do not use this entry point for a real payment system.	PolicyCenter uses the exit point to transfer control to the external payment system.
3.	After you enter a payment, the demonstration payment collection page uses the PCF exit point called <code>FinishPaymentInstrument</code> . Do not use this exit point for a real payment system.	You enter a new payment instrument in the external payment system.
4.	The <code>FinishPaymentInstrument</code> exit point calls the PolicyCenter PCF entry point called <code>JobWithNewPaymentInstrument</code> .	The payment system navigates to PolicyCenter by using the PolicyCenter PCF entry point <code>JobWithNewPaymentInstrument</code> .
7.	PolicyCenter uses the registered billing system plugin (<code>IBillingSystemPlugin</code>) implementation to notify the billing system of the new payment instrument. PolicyCenter calls the plugin method <code>addPaymentInstrumentTo</code> .	

See also

- “Configure PolicyCenter to use a real payment system” on page 549

Configure PolicyCenter to use a real payment system

Perform these steps to configure PolicyCenter to take customer payments from a real payment system rather than the demonstration system that the base configuration provides.

About this task

For demonstration purposes only, in the base configuration, the `externalPaymentLocation` method on the `gw.billing.PolicyPeriodBillingInstructionsManager` object calls a demonstration version of a payment system that PolicyCenter itself implements.

To integrate with a real payment system, you must make the following changes to PolicyCenter.

Procedure

1. In your application `config.xml` file, find the parameter `PaymentSystemURL`. Set that parameter to the URL of your real payment system.
2. Change the code for the `externalPaymentLocation` method to contact your real payment system:
 - a. Locate the line that looks like the following code. Note the commented out URL:


```
var paymentURL = baseURL //gw.api.system.PCConfigParameters.PaymentSystemURL.Value
```
 - b. Change the assignment to use the commented-out code:


```
var paymentURL = gw.api.system.PCConfigParameters.PaymentSystemURL.Value
```
3. If necessary, change the code for the `externalPaymentLocation` method to return to the required PolicyCenter location.
 - a. Locate the line that looks like the following code. Note the commented out URL:


```
var returnUrl = baseURL
// gw.plugin.Plugins.get(gw.plugin.webconfig.IPolicyCenterWebConfigPlugin).PolicyCenterURL
```
 - b. Change the assignment instead to use the commented-out code:


```
var returnUrl =
gw.plugin.Plugins.get(gw.plugin.webconfig.IPolicyCenterWebConfigPlugin).PolicyCenterURL
```
4. Configure the entry point `JobWithNewPaymentInstrument` to send any additional attributes that the real payment system sends to PolicyCenter.

Accessing an external payment gateway to add an up-front payment

The following steps show the tasks that you perform and the actions that the base configuration of PolicyCenter takes to use a payment gateway to create an electronic up-front payment:

1. You start a submission, change, renewal, or rewrite job.
 2. You quote the policy and navigate to the **Payment** screen before binding the policy. PolicyCenter displays billing methods, payment methods, and associated installment or reporting plans retrieved from the billing system.
 3. You select a billing method and payment schedule, and then preview payment plans retrieved from the billing system. This part of the wizard is similar for all lines of business. The PCF file `UpFrontPaymentDV.Editable.pcf` provides the user interface for the up-front payments. This PCF file lists up-front payments and provides options for adding one or more up-front payments. You select **Electronic Payment** to initiate an up-front payment that uses a payment gateway. The down-payment amount appears. You can change this value.
 4. PolicyCenter uses the registered billing system plugin (`IBillingSystemPlugin`) implementation to request the known payment instruments from the billing system. PolicyCenter calls its plugin method `getExistingPaymentInstruments`.
 - To add a payment on a new payment instrument, in `UpFrontPaymentDV.Editable`, in **Choose Card**, you click **Other**.
 - To add a payment on a known payment instrument, in `UpFrontPaymentDV.Editable`, in **Choose Card**, you click the value that corresponds to the card to use. Typically, the value does not show the whole credit card number.
 To process the card and payment, click the **Take Payment** button.
 5. The action that the **Take Payment** button takes is to call a method called `createElectronicPayment` on the `gw.billing.PolicyPeriodBillingInstructionsManager` object that the page uses. The `PolicyPeriodBillingInstructionsManager` class delegates the implementation of this method to the `gw.billing.UpFrontPaymentStarterContainerImpl` class.
- If you select **Save for Future Use**, PolicyCenter retains the information about the new payment instrument and sends that information to the billing system. PolicyCenter uses the registered billing system plugin

(`IBillingSystemPlugin`) implementation to notify the billing system of the new payment instrument. PolicyCenter calls the plugin method `addPaymentInstrumentTo`. On successful completion of a payment, the **Payment** screen displays information about the card in the list of known payment instruments. This information typically includes the last four digits of the credit card number.

6. The `createElectronicPayment` method creates a `PaymentInformation` Gosu object, which holds information about the payment. This object is suitable for use as an argument to web service methods. Next, the `createElectronicPayment` method calls the `takeElectronicPayment` method. The `takeElectronicPayment` method tests the value of the `RedirectToPaymentGateway` property on the `PaymentGatewayConfig` plugin. If the value of the `RedirectToPaymentGateway` property is `false`, electronic payment is not available.
7. The `takeElectronicPayment` method uses the registered payment gateway plugin (`PaymentGatewayPlugin`) methods to access the payment gateway. Typically, the payment gateway is on a separate physical computer across the Internet. The `takeElectronicPayment` method uses different plugin methods for a new payment instrument and a known payment instrument:
 - For a new payment instrument, `takeElectronicPayment` calls the `requestSecureToken` plugin method to receive a token that identifies the payment. After saving a `PaymentGatewayTransaction` entity to the database, `takeElectronicPayment` calls the `redirectToPaymentGateway` plugin method, which navigates to the payment gateway user interface. The payment gateway plugin processes the response from the payment gateway. If you select **Save for Future Use**, you must retrieve tokenized information to identify the credit card from the payment gateway.
 - For a known payment instrument, PolicyCenter already has tokenized information that identifies the payment instrument to the payment gateway. The `takeElectronicPayment` method calls the `takePaymentUsingPaymentInstrument` plugin method, which uses the tokenized information. After saving a `PaymentGatewayTransaction` entity to the database, `takeElectronicPayment` processes the response from the payment gateway.

Redirection to the payment gateway for a new payment instrument occurs differently for the demonstration gateway and a real gateway:

- For demonstration purposes, the `redirectToPaymentGateway` plugin method invokes the PCF exit point called `DemoPaymentGatewayPage`, which calls a demonstration version of a payment gateway that PolicyCenter itself implements.
- For a production environment, the code for the `redirectToPaymentGateway` plugin method uses a different PCF exit point to open the user interface for your real payment gateway.

The following table compares the two payment gateways:

Step	Demonstration payment gateway	Real payment gateway
1.	The <code>redirectToPaymentGateway</code> plugin method passes the transaction amount and job number to the demonstration payment system, which PolicyCenter implements as PCF pages and Gosu objects.	The <code>redirectToPaymentGateway</code> plugin method exits to the real payment system. The PCF page <code>PaymentGateway.pcf</code> is the exit point to the payment gateway.
2.	The PCF page <code>DemoPaymentGatewayScreen.pcf</code> opens and emulates a the user interface for providing credit card details in a payment gateway.	PolicyCenter uses the exit point to transfer control to the external payment system.
3.	After you enter the credit card details and click the Take Payment button, the demonstration payment gateway page validates the values on the page and opens the <code>DemoTransactionConfirmationPage.pcf</code> page.	You enter the credit card and payment details in the external payment system. The gateway typically does a postback action on completion. A PolicyCenter servlet processes this action. The gateway typically displays a payment confirmation page after the payment completes.

Step	Demonstration payment gateway	Real payment gateway
4.	When you click the Return to Merchant Site link on the <code>DemoTransactionConfirmationPage.pcf</code> page, the page transfers control to the <code>PaymentGatewayForward.pcf</code> page. The <code>PaymentGatewayForward.pcf</code> page builds a <code>PaymentGatewayResponseBase</code> Gosu object, and then forwards control to the <code>processPaymentResponseAndRedirect</code> method of a new <code>PaymentGatewayResponseProcessor</code> Gosu object. This method adds the payment to the <code>Job</code> entity instance.	If the payment gateway performs a postback action, the <code>PaymentGatewaySilentPostServlet</code> uses the <code>mapSilentPostRequestToPaymentGatewayResponse</code> plugin method to map the <code>HttpServletRequest</code> object from the external payment gateway to a <code>PaymentGatewayResponseBase</code> Gosu object. The servlet then calls the <code>processPostbackPaymentResponse</code> method of a new <code>PaymentGatewayResponseProcessor</code> Gosu object. This method adds the payment to the <code>Job</code> entity instance. If the payment gateway does not perform a postback action, the payment gateway entry point to PolicyCenter processes the response.
5.	No payment message is sent to the billing system.	PolicyCenter sends information about the payment to the billing system. If the billing system is <code>BillingCenter</code> , PolicyCenter calls the <code>PaymentAPI</code> method that creates a suspense payment.
6.	Control returns to the Payment screen.	Control returns to the Payment screen.
8.	For a successful or voided payment, PolicyCenter removes the <code>PaymentGatewayTransaction</code> entity from the database.	
9.	You issue or bind the policy. PolicyCenter uses the registered billing system plugin (<code>IBillingSystemPlugin</code>) implementation to send a Gosu object to the billing system. PolicyCenter notifies the billing system that it has received a payment by setting the <code>OfferNumber</code> property on the Gosu object. The value of the <code>OfferNumber</code> property is the <code>JobNumber</code> property of the <code>Job</code> .	

See also

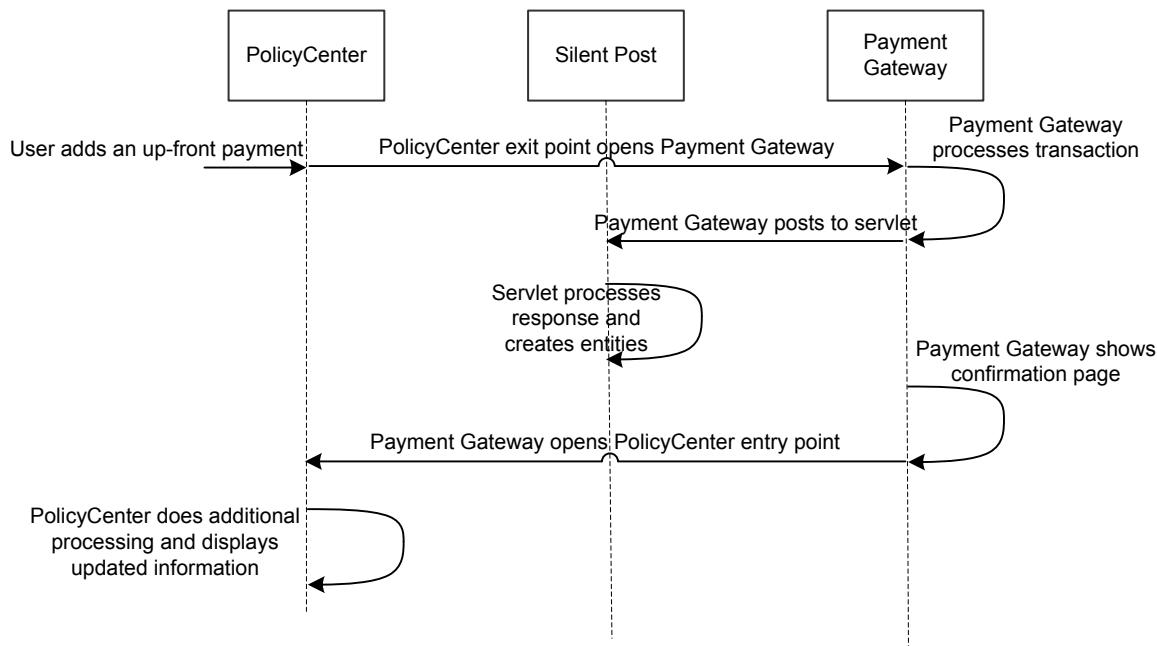
- “Configuring PolicyCenter to use a real payment gateway” on page 552

Configuring PolicyCenter to use a real payment gateway

The following illustration shows a simplified representation of the typical information flow between PolicyCenter and the payment gateway:

Up-front Payments using a Payment Gateway

Typical Flow



PolicyCenter components for communication with a payment gateway

The following sections describe the PolicyCenter components that support communication with a payment gateway.

Payment gateway plugin components

The following table describes the plugins that support communication with a payment gateway.

Plugin	Description
Payment gateway configuration plugin	<p>Purpose</p> <p>Specifies key information about a payment gateway, such as the URL.</p> <p>Class or interface</p> <p>Interface: <code>gw.plugin.paymentgateway.PaymentGatewayConfigPlugin</code></p> <p>Demonstration implementation: <code>gw.plugin.paymentgateway.DefaultPaymentGatewayConfig Plugin</code></p> <p>See also</p> <ul style="list-style-type: none"> “Payment gateway configuration plugin” on page 163 “Implementing the payment gateway configuration plugin” on page 556
Payment gateway plugin	<p>Purpose</p> <p>Communicates with the payment gateway, sending payments, and receiving responses.</p> <p>Class or interface</p> <p>Interface: <code>gw.plugin.paymentgateway.PaymentGatewayPlugin</code></p> <p>Demonstration implementation: <code>gw.plugin.paymentgateway.StandAlonePaymentGatewayPlugin</code></p> <p>See also</p> <ul style="list-style-type: none"> “Payment gateway plugin” on page 164 “Implementing the payment gateway plugin” on page 556

The following table describes the servlet that supports communication with a payment gateway.

Servlet	Description
Payment gateway silent post servlet	<p>Purpose Processes a postback action from the payment gateway after the gateway processes a payment. This servlet ensures that PolicyCenter and the billing system receive information about a payment even if the user closes the browser.</p> <p>Class Demonstration implementation: <code>gw.payment.paymentgateway.PaymentGatewaySilentPost Servlet</code></p> <p>See also “Implementing the silent post servlet for the payment gateway” on page 558</p>

Payment gateway PCF file components

The following table describes the navigation and user-interface PCF files that support communication with a payment gateway.

PCF file	Description
Exit point from PolicyCenter to the payment gateway	<p>Purpose Maps variables to URL parameters and transfers control to the payment gateway.</p> <p>File Demonstration implementation: <code>PaymentGateway.pcf</code></p> <p>See also “Implementing the payment gateway exit point” on page 559</p>
Entry point to PolicyCenter from the payment gateway	<p>Purpose Maps URL parameters to location parameters and passes control to the navigation forward PCF.</p> <p>File Demonstration implementation: <code>PaymentGatewayEntryPoint.pcf</code></p> <p>See also “Implementing the payment gateway entry point” on page 559</p>
Navigation forward to PolicyCenter location	<p>Purpose Performs an action that processes the parameters from the entry point, and then redirects to the appropriate PolicyCenter location.</p> <p>File Demonstration implementation: <code>PaymentGatewayForward.pcf</code></p> <p>See also “Implementing the payment gateway entry point” on page 559</p>
User interface for adding up-front payments for a job	<p>Purpose Provides a list of the available types of payment instrument, a list of the available electronic payment instruments, suggests an amount to pay, and communicates with the payment gateway.</p> <p>File <code>UpFrontPaymentDV.Editable.pcf</code></p> <p>See also “Using and controlling the up-front payment user interface” on page 558</p>

Payment gateway state control components

The following table describes the Gosu classes and interfaces that support state control of communication with a payment gateway.

Gosu component	Description
Payment method and payment plan manager	<p>Purpose</p> <p>Tracks the payments that the selected payment plan requires and controls the information that appears on the Up-front Payment user interface.</p> <p>Class or interface</p> <p>Interfaces: All delegated to other implementation classes.</p> <p>Demonstration implementation: <code>gw.billing.PolicyPeriod BillingInstructionsManager</code></p> <p>See also</p> <p>“Using and controlling the up-front payment user interface” on page 558</p>
Up-front payments manager	<p>Purpose</p> <p>Tracks the up-front payments that the user enters.</p> <p>Class or interface</p> <p>Interface: <code>gw.billing.UpFrontPayment StateContainer</code></p> <p>Demonstration implementation: <code>gw.billing.UpFrontPayment StateContainerImpl</code></p> <p>See also</p> <p>“Using and controlling the up-front payment user interface” on page 558</p>
Payment gateway response processor	<p>Purpose</p> <p>Processes the information in a response object that implements the <code>PaymentGatewayResponse</code> interface. The <code>PolicyPeriod BillingInstructionsManager</code> object uses controls .</p> <p>Class or interface</p> <p>Demonstration implementation: <code>gw.payment.paymentgateway .PaymentGatewayResponse Processor</code></p> <p>See also</p> <p>“Implementing payment gateway data transfer” on page 559</p>

Payment gateway data transfer components

The following table describes the Gosu classes and interfaces that support data transfer in communication with a payment gateway.

Gosu component	Description
Tokenized basic information about a credit card	<p>Purpose</p> <p>Provides two <code>String</code> values. One value, which typically has a format similar to <code>xxxx-1234</code>, identifies a credit card on the Up-front Payment user interface. The other value is a token that identifies the card to the payment gateway.</p> <p>Class or interface</p> <p>Interface: <code>gw.api.payment.paymentgateway.Payment InstrumentDetails</code></p> <p>Demonstration implementation: <code>gw.payment.paymentgateway.impl.TokenizedCreditCardDetails</code></p>
Information that links an electronic payment instrument and the policy to which it applies	<p>Purpose</p> <p>Provides basic information about each payment instrument for which the billing system is the system of record. The <code>PolicyPeriodBillingInstructionsManager</code> object uses these objects in communication with the external payment gateway.</p>

Gosu component	Description
	<p>Class or interface Interface: <code>gw.plugin.billing.BillingPaymentInstrument</code> Demonstration implementation: <code>gw.plugin.billing.BillingPaymentInstrumentImpl</code></p>
Information about a payment to process	<p>Purpose Transfers information about a payment to the payment gateway for processing.</p> <p>Class or interface Interface: <code>gw.api.payment.paymentgateway.PaymentInformation</code> Demonstration implementation: <code>gw.payment.paymentgateway.impl.PaymentInformationBase</code></p>
Information from the payment gateway	<p>Purpose Contains the information that a payment gateway returns from a request that PolicyCenter makes.</p> <p>Class or interface Interface: <code>gw.api.payment.paymentgateway.PaymentGatewayResponse</code> Demonstration implementation: <code>gw.payment.paymentgateway.impl.PaymentGatewayResponseBase</code></p>

See also

- “Using and controlling the up-front payment user interface” on page 558
- “Implementing payment gateway data transfer” on page 559

Implementing the payment gateway configuration plugin

The payment gateway configuration plugin provides basic merchant information that PolicyCenter uses for communication with the payment gateway and information about how to access the payment gateway. PolicyCenter provides a demonstration implementation of the payment gateway configuration plugin in `gw.plugin.paymentgateway.DefaultPaymentGatewayConfigPlugin`. This plugin calls a demonstration implementation of the payment gateway that PolicyCenter itself implements. PolicyCenter wraps calls to the payment gateway configuration plugin in the `gw.payment.paymentgateway.PaymentGatewayConfig` class. Create your own plugin class to implement the `PaymentGatewayConfigPlugin` interface. If the value that your plugin returns for the `RedirectToPaymentGateway` configuration parameter is false, PolicyCenter does not take electronic payments. Modify `PaymentGatewayConfig.gs` to perform postprocessing of the configuration parameter values and change the plugin registry of the `PaymentGatewayConfigPlugin` interface to use your plugin class.

Implementing the payment gateway plugin

For demonstration purposes only, the base configuration of PolicyCenter provides an implementation of the `PaymentGatewayPlugin` interface in `gw.plugin.paymentgateway.StandAlonePaymentGatewayPlugin`. Various plugin methods in this class return hard-coded values rather than calling the payment gateway to retrieve information. This class does not use any of the exit point from PolicyCenter to the payment gateway, the entry point that returns from the payment gateway, or the silent post servlet. In this demonstration plugin class, the `redirectToPaymentGateway` plugin method calls a demonstration version of a payment system that PolicyCenter itself implements as PCF files. These files are located in the Page Configuration folder `pcf/payment/demo`.

Create your own plugin class to implement the `PaymentGatewayPlugin` interface. Change the plugin registry of the `PaymentGatewayConfigPlugin` interface to use your plugin class. Your plugin class must be able to:

- Take information about the payment amount and electronic payment instrument from the **Payment** screen in the PolicyCenter user interface. Save the payment instrument if the user selects **Save for Future Use**.
- Send the payment information to an external payment gateway in the form that the gateway requires.
- Retrieve success or failure information about a payment request from the payment gateway. Save information about a successful request on the `Job` entity instance for the policy period. Send information about a successful payment or credit to the billing system.
- Ensure that a payment request does not get processed multiple times. Recognize that the **Up-front Payment** user interface does not show a successful payment and synchronize the user interface to show the result of the payment gateway transaction.
- Use the payment gateway to reverse a payment, and then remove the reversed payment from the `Job` entity instance.

Taking information about a payment

PolicyCenter uses a Gosu object that implements the `PaymentInformation` interface to hold information about the up-front payment to send to the payment gateway. This object includes information about the policy period, the payment instrument, and the amount to pay.

The **Payment** screen retrieves information about known payment instruments from the system of record, which is usually the billing system. This information is in the form of Gosu objects that implement the `BillingPaymentInstrument` interface. Each object includes a `String` token that identifies the credit card to the payment gateway. For a new payment instrument, if the user selects **Save for Future Use**, you must call the `getCardDetails` plugin method to get a new token from the payment gateway. This method takes a `String` that identifies the credit card as an argument and returns a `TokenizedCreditCardDetails` object.

The **Payment** screen also provides the amount to pay as an up-front payment.

Sending payment information to the payment gateway

PolicyCenter uses a Gosu object that implements the `PaymentInformation` interface to transfer data to the payment gateway plugin. PolicyCenter saves a `PaymentGatewayTransaction` entity instance before sending a payment request to the payment gateway. This transaction has a `String Reference` property that identifies the payment gateway request. Typically, you implement separate paths for taking payment from a known payment instrument and from a new payment instrument.

For a known payment instrument, call the `takePaymentUsingPaymentInstrument` plugin method. This method takes an object that implements the `PaymentInformation` interface as an argument and returns an object that implements the `PaymentGatewayResponse` interface. The payment gateway does not need to present a user interface because it has all the necessary information to take the payment. The payment information argument includes the token that identifies the credit card to the payment gateway as well as the amount to charge to the card. If the payment request succeeds, the response processor creates an `UpFrontPayment` entity instance for the payment and updates the set of up-front payments on the job.

For a new payment instrument, call the `requestSecureToken` and `redirectToPaymentGateway` plugin methods. The `requestSecureToken` method takes an object that implements the `PaymentInformation` interface as an argument and returns an object that implements the `PaymentGatewayResponse` interface. For example, depending on your payment gateway, in the `requestSecureToken` method, you use the payment information to construct an HTTP GET command and receive the token in the response. The `redirectToPaymentGateway` method takes the response object that the `requestSecureToken` method returns and an object that implements the `PaymentInformation` interface as arguments. Use the payment information argument to specify any payment details that are not encoded in the secure token in the response argument. This method opens the exit point that transfers control to the payment gateway.

Some payment gateways require verification of a new payment instrument. To provide this verification, implement the `submitAccountVerification` plugin method. This method takes an object that implements the `PaymentInformation` interface as an argument and returns an object that implements the `PaymentGatewayResponse` interface.

The payment gateway typically uses a servlet to send the result of the payment request that uses a new payment instrument to PolicyCenter.

Retrieving result information from the payment gateway

PolicyCenter uses a Gosu object that implements the `PaymentGatewayResponse` interface to receive data from the payment gateway. Implement the `isApproved` and `isVoided` plugin methods to check the status of the payment request. Both of these methods take an object that implements the `PaymentGatewayResponse` interface as an argument and return a `boolean` value.

A servlet typically processes the payment information from the payment gateway user interface.

Ensuring synchronization between PolicyCenter and the payment gateway

The standard processing of a successful request to the payment gateway deletes the `PaymentGatewayTransaction` entity instance that identifies the request after updating the up-front payment information on the job. The `Payment` screen updates the list of payments and the outstanding payment information. On entry to the `Payment` screen, PolicyCenter checks for existing `PaymentGatewayTransaction` entity instances and displays a **Synchronize** button and a message requesting synchronization. The action that the **Synchronize** button takes is to call the `inquiryPaymentGatewayTransaction` plugin method. Implement this method to check the status of the payment request that relates to the `PaymentGatewayTransaction` entity instance. This method takes the `Reference` property of the transaction as a `String` argument and returns an object that implements the `PaymentGatewayResponse` interface. The response processor updates the `PaymentGatewayTransaction`, `Job`, and `UpFrontPayment` entity instances.

Reversing a payment

The user can choose to remove an up-front payment from a job. This action credits the payment on the payment gateway. Similarly to processing a payment, PolicyCenter uses a Gosu object that implements the `PaymentInformation` interface to hold information about the up-front payment to reverse. Implement the `refundFullAmountForTransaction` plugin method to credit the previous payment. This method takes an object that implements the `PaymentInformation` interface as an argument and returns an object that implements the `PaymentGatewayResponse` interface. If the credit request succeeds, the response processor removes the `UpFrontPayment` entity instance for the payment and updates the set of up-front payments on the job.

Using and controlling the up-front payment user interface

The `UpFrontPaymentDV.Editable.pcf` file provides the user interface for managing up-front payments before you bind or issue a policy. This file uses Gosu objects for state control and data transfer. The state control objects call payment gateway configuration plugin methods and payment gateway plugin methods. The payment gateway plugin uses the data transfer objects to provide information to and receive information from the payment gateway.

The `gw.billing.PolicyPeriodBillingInstructionsManager` Gosu class provides overall state control for the `Payment` screen. This class delegates control of individual areas of the screen to other classes. For state control of up-front payments, the `gw.billing.UpFrontPaymentStateContainerImpl` class implements the `gw.billing.UpFrontPaymentStateContainer` interface. Modify these two classes to implement the requirements of communicating with your payment gateway. Typically, you modify the methods in `UpFrontPaymentStateContainerImpl` that call the payment gateway plugin methods. These methods are `takeElectronicPayment`, `synchronizePendingPaymentGatewayTransactions`, and `removePayment`.

Implementing the silent post servlet for the payment gateway

A typical payment gateway uses a *postback* action to process the response for a payment that you take from an electronic payment instrument. This postback action is an HTTP POST command that sends the result of the payment request to the application that calls the payment gateway, which is PolicyCenter. The payment gateway calls the servlet and typically displays a page that shows the result of the request. PolicyCenter uses a servlet that has no user interaction to process the postback. Using a servlet ensures that processing of the electronic payment does not get lost if the user closes the browser and that payment occurs only once. The `servlet/servlets.xml` file identifies the servlet class.

PolicyCenter provides a demonstration implementation of the payment gateway silent post servlet in the `gw.payment.paymentgateway.PaymentGatewaySilentPostServlet` class. This servlet calls the `PaymentGatewayPlugin` plugin method `mapSilentPostRequestToPaymentGatewayResponse` to create a `gw.payment.paymentgateway.PaymentGatewayResponse` object from the `HttpServletRequest` object from the payment gateway. The servlet then uses the PolicyCenter response processor to create an `UpFrontPayment` entity instance that stores information about the payment. The response processor updates the set of up-front payments on the job.

Create your own servlet or change the implementation of the `doPost` method in the demonstration servlet to process the `HttpServletRequest` from your payment gateway. If you create your own servlet, include its definition in the `servlets.xml` file. Configure your payment gateway to use the URL to this servlet to perform the postback.

Implementing the payment gateway exit point

The payment gateway exit point maps payment information from PolicyCenter to URL parameters on the URL that accesses the payment gateway. PolicyCenter provides a demonstration implementation of the payment gateway exit point in the `PaymentGateway.pcf` file. This implementation uses the following parameters to build a URL to access the payment gateway:

- `paymentGatewayURL` – The URL for the payment gateway. Use the payment gateway configuration plugin to set up this URL.
- `secureToken` – Encoded information that identifies an individual transaction. Encoding the information ensures that third parties cannot examine or modify the information during its transfer from PolicyCenter to the payment gateway. The `requestSecureToken` plugin method takes an object that implements the `PaymentInformation` interface as an argument and returns an object that implements the `PaymentGatewayResponse` interface. This response object provides the value for `secureToken`. Typically, a secure token expires after a length of time or after a number of unsuccessful attempts to complete the payment. You specify these parameters when you set up the payment gateway.
- `secureTokenID` – A value that you provide to identify the `secureToken` to the payment gateway.

Implement the `PaymentGatewayPlugin` plugin method `redirectToPaymentGateway` to use the exit point. In the exit point, map PolicyCenter values to URL parameters that your payment gateway requires.

Implementing the payment gateway entry point

The payment gateway entry point takes the URL parameters from the payment gateway URL that returns control to PolicyCenter and maps those parameters to a payment gateway response object. PolicyCenter provides a demonstration implementation of the payment gateway entry point in the `PaymentGatewayEntryPoint.pcf` file. This demonstration entry point transfers control to the `PaymentGatewayForward.pcf` file that processes the response and transfers control to the PolicyCenter user interface. This implementation builds the following parameters to build a URL to access the payment gateway:

The demonstration implementation of the payment gateway calls the `PaymentGatewayForward.pcf` file directly and does not use the entry point.

Edit the entry point file to use the URL parameters that your payment gateway provides. Edit the `PaymentGatewayForward.pcf` file to convert values from the entry point to a payment gateway response object and transfer control to the PolicyCenter user interface.

Implementing payment gateway data transfer

PolicyCenter uses multiple data transfer objects (DTO) to transfer information between the payment gateway and PolicyCenter and between PolicyCenter and the billing system. The DTO classes each implement an interface that defines the DTO properties.

See also

- “Implementing the payment gateway plugin” on page 556
- “Payments Web Service APIs (PaymentAPI)” in *BillingCenter Integration Guide*

Implementing a tokenized credit card class

PolicyCenter uses a Gosu object that represents a tokenized credit card to receive information about a new credit card from the payment gateway. Your billing system is typically the system of record for this information for known credit cards. The payment gateway response processor calls the `PaymentGatewayPlugin` plugin method `getCardDetails` to receive the tokenized credit card object for a new credit card. This object maps the `String` token that the payment gateway uses to identify the card with a `String` value that the **Payment** screen displays to the user. Typically, the value that the **Payment** screen displays has a format that is similar to xxxx-1234. The Gosu class implements the `gw.api.payment.paymentgateway.PaymentInstrumentDetails` interface. This interface provides two `String` properties:

- `CardName` – The value to display on the user interface
- `Token` – The value that identifies the credit card to the payment gateway

The `gw.payment.paymentgateway.impl.TokenizedCreditCardDetails` class is a demonstration implementation of the tokenized credit card. Edit this class or create your own implementation to support the requirements of your payment gateway and billing system.

Implementing a payment instrument class

PolicyCenter uses a Gosu object that represents a credit card to send information to the payment gateway plugin and to transfer information to and from the billing system. Your billing system is typically the system of record for this information for known credit cards. The `PaymentGatewayPlugin` plugin method `getCardDetails` provides the tokenized credit card object to the payment gateway response processor, which transforms that object to a new payment instrument. If the user chose to save the credit card for future use, the response processor sends the payment instrument object to the billing system. The **Payment** screen retrieves payment instruments from the billing system and displays a `String` value to identify each payment instrument. The user selects a payment instrument or **Other** to specify a new payment instrument. The Gosu class implements the `gw.plugin.billing.BillingPaymentInstrument` interface. This interface provides the following properties:

- `PublicID` – The `String` identifier of the payment instrument in the system of record
- `PaymentMethod` – The `AccountPaymentMethod` typekey that specifies the payment method for the policy, which is either installments or reporting
- `DisplayName` – The `String` value to display on the user interface
- `OneTime` – The `Boolean` value that specifies whether this instrument is for use for an individual payment only
- `Token` – The `String` value that identifies the credit card to the payment gateway
- `Detail` – An optional `String` value to store arbitrary information that your systems require

The `gw.plugin.billing.BillingPaymentInstrumentImpl` class is a demonstration implementation of the payment instrument. Edit this class or create your own implementation to support the requirements of your payment gateway and billing systems.

Implementing a payment information class

PolicyCenter uses a Gosu object to encapsulate information about a payment. PolicyCenter uses this object to send information about a payment or credit to the payment gateway plugin and the billing system. This object includes information about the billing account, the policy period, the payment instrument, and the amount to pay. You cannot use this object as an argument to a web service because it includes a reference to a `PolicyPeriod` entity. The payment gateway plugin must transform this object into an object that identifies the policy period by another value, such as the job number. The `UpFrontPaymentStateContainerImpl` method `takeElectronicPayment` creates a payment information object to use as an argument to the payment gateway plugin methods `takePaymentUsingPaymentInstrument`, `requestSecureToken`, and `redirectToPaymentGateway`. The `UpFrontPaymentStateContainerImpl` method `removePayment` creates a payment information object to use as an argument to the payment gateway plugin method `refundFullAmountForTransaction`. The Gosu class implements

the `gw.api.payment.paymentgateway.PaymentInformation` interface. This interface provides the following properties:

- `TransactionAmount` – The `BigDecimal` value to pay or credit.
- `TransactionCurrency` – The `Currency` value to use for the payment or credit.
- `BillingAccountNumber` – The `String` value that identifies the account in the billing system.
- `PolicyPeriod` – The `PolicyPeriod` to which the payment applies.
- `PaymentInstrument` – The `String` value that the user interface displays for the payment instrument
- `TransactionTypeCode` – The `PaymentTransactionType` value for installments or reporting.
- `TransactionID` – The `String` value that identifies the payment transaction, if required by the payment gateway or the billing system.
- `SaveForFutureUse` – The `boolean` value that specifies whether to store the payment instrument to make it available on the user interface.
- `InternalPaymentReference` – The `String` value that identifies the policy period to which the payment or credit applies. For integration with BillingCenter, set this property to the job number.

The `gw.payment.paymentgateway.impl.PaymentInformationBase` class is a demonstration implementation of the payment instrument. Edit this class or create your own implementation to support the requirements of your payment gateway and billing system. If BillingCenter is your billing system, use the payments web service API provided by PaymentAPI to manage suspense payments. Create a suspense payment for a successful payment transaction. Remove a suspense payment for a successful credit transaction.

Implementing a payment gateway response class

PolicyCenter uses a Gosu object that represents a response from the payment gateway to track the result of any request to the gateway. Many methods in the payment gateway plugin use a payment gateway response object as either an argument or a return value. The up-front payment state container, the silent post servlet, and the payment gateway forward page in the demonstration implementation of a payment gateway use or create these objects. The Gosu class implements the `gw.api.payment.paymentgateway.PaymentGatewayResponse` interface. This interface provides the following read-only properties:

- `Result` – The `String` value of the code that the payment gateway returns, for example, “0” for success
- `ReferenceNumber` – The `String` value that identifies the transaction
- `ResponseMessage` – The `String` value that the payment gateway returns, for example, “Approved”
- `Amount` – The `String` value that the payment gateway charged to the credit card
- `Token` – The `String` value that contains the secure token for the transaction
- `OriginalTransactionID` – The `String` value equal to the `ReferenceNumber` of a previous transaction for crediting or checking the status of that transaction
- `TransactionType` – The `PaymentTransactionType` value for installments or reporting

The `gw.payment.paymentgateway.impl.PaymentGatewayResponseBase` class is a demonstration implementation of the payment gateway response. Edit this class or create your own implementation to support the requirements of your payment gateway.

Implementing a payment gateway response processor class

PolicyCenter uses a Gosu object to process the result of a request to the payment gateway. The up-front payment state container, the silent post servlet, and the payment gateway forward page in the demonstration implementation of a payment gateway create these objects. The response processor saves transaction information as a `PaymentGatewayTransaction` entity instance before PolicyCenter requests a payment from the payment gateway. On processing a response for a successful payment request, the response processor creates an `UpFrontPayment` entity instance and removes this `PaymentGatewayTransaction` entity instance.

The `gw.payment.paymentgateway.PaymentGatewayResponseProcessor` class is a demonstration implementation of a payment gateway response processor. The constructor takes a payment response object and optionally, a payment information object as arguments. This class provides the following methods:

- `processPaymentResponse` – Used by the up-front payment state container to process the response from a payment or credit request
- `processPaymentResponseAndRedirect` – Used by the payment gateway forward page to send control to the applicable user interface
- `processPostBackPaymentResponse` – Used by the silent post servlet to add a successful payment to the job
- `processSynchronizedPaymentResponse` – Used by the up-front payment state container to synchronize payments that the payment gateway has processed, but which do not appear on the **Payment** screen

Your implementation of a response processor also sends information about the payment to the billing system as a suspense payment. For example, if the billing system is `BillingCenter`, the response processor calls the `PaymentAPI` method that creates a suspense payment. The demonstration implementation of the response processor does not send this payment information to the billing system.

Part 8

Contact integrations

Contact integration

This topic discusses how to integrate PolicyCenter with an external contact management system other than ContactManager, and also covers configuring other actions relating to contacts.

PolicyCenter includes built-in support for integrating with Guidewire ContactManager.

See also

- *Guidewire Contact Management Guide*

Integrating with a contact management system

PolicyCenter integrates with an external contact management system through two plugin interfaces. Each one has a different function.

ContactSystemPlugin

Integrate your external contact management system with PolicyCenter.

IContactConfigPlugin

Configure how PolicyCenter handles contacts.

The most important contact-related plugin interface is the contact system plugin **ContactSystemPlugin**.

Integrate with an external contact management system by registering a class that implements the **ContactSystemPlugin** plugin interface.

For use with internal contacts only, PolicyCenter includes the following **ContactSystemPlugin** plugin interface implementation.

```
gw.plugin.contact.impl.StandAloneContactSystemPlugin
```

The PolicyCenter base configuration supports integration with Guidewire ContactManager™. If you have a Client Data Management module license and install ContactManager, do not write your own implementation of the **ContactSystemPlugin** plugin interface. Instead, use the plugin implementation `gw.plugin.contact.ab1000.ABContactSystemPlugin`.

The PolicyCenter base configuration has built-in support for integration with Guidewire ContactManager. If you have a license for the optional Client Data Management module, you can install and integrate ContactManager.

Your implementation of the contact system plugin defines the contract between PolicyCenter and the code that initiates requests to an external contact management system. PolicyCenter 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.

PolicyCenter 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 PolicyCenter needs to retrieve a contact from the contact management system, it calls the `retrieveContact` method of this plugin. The plugin takes only one argument, the unique ID of the contact. Your plugin implementation needs to get the contact from the external system by using whatever network protocol is appropriate, and then populate a contact record object.

For this plugin, the contact record object that you must populate is a `Contact` entity instance.

If you need the PolicyCenter 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 PolicyCenter, you must modify the contact-related PCF files to extract and display the new field.

Inbound contact integrations

For inbound integration, PolicyCenter publishes a WS-I web service called `ContactAPI`. This web service enables an external contact management system to notify PolicyCenter of updates, deletes, and merges of contacts in that external system. When PolicyCenter receives a notification from the external contact management system, PolicyCenter can update its local copies of those contacts to match.

See also

- “Contact web service APIs” on page 573

Asynchronous messaging with the contact management system

When a user in PolicyCenter creates or updates a local `Contact` instance, PolicyCenter 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`. PolicyCenter calls the message transport plugin implementation to send a message.
 - A message request (`MessageRequest`) plugin implementation called `ContactMessageRequest`. PolicyCenter calls the message request plugin to update the message payload immediately before sending the message if necessary.

In the default configuration, PolicyCenter 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, PolicyCenter sends the message to the messaging destination. There are two phases of this process.
 - a. PolicyCenter 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. PolicyCenter calls the `beforeSend` method of the `ContactMessageRequest` plugin to update the payload before the message is sent.
 - b. PolicyCenter 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.

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 PolicyCenter, create a new instance of a contact in the current transaction's bundle.
 - a. Create the new instance by using the instance of `ContactCreator` that your plugin method gets as a method argument. This class standardizes finding and creating contacts within PolicyCenter. It has a `loadOrCreateContact` method that you can use for creating a contact.
 - b. Populate the new contact entity instance with information from your external contact.

Your code might look like something like the following example.

```
override function retrieveContact(addressBookUID : String, creator : ContactCreator) : Contact {  
    var returnedContact : Contact = null  
    var contactXml = retrieveContactXML(addressBookUID)  
    if (contactXml != null) {  
        var contactType =  
            _mapper.getNameMapper().getLocalEntityName(contactXml.EntityType)
```

```

        returnedContact = creator.loadOrCreateContact(contactXml.LinkID, contactType)

        validateAutoSyncState(returnedContact, addressBookUID)
        overwriteContactFromXml(returnedContact, contactXml)
    }
    return returnedContact
}

```

5. Return that object as the result from your `retrieveContact` method.

Contact searching

To support contact searching, your plugin must respond to a search request from PolicyCenter. PolicyCenter 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`
- `OfficialId`
- `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.

Properties related to proximity search are unsupported. For example, you cannot search by latitude or longitude using this API.

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 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 `ABCContactAPISearchResult` Gosu class in ContactManager. This class is in the package `gw.webservice.ab.ab1000.abcontactapi`.

See also

- *Guidewire Contact Management Guide*

Support for finding duplicate contacts

Your plugin must tell PolicyCenter whether the external contact management system supports finding duplicates. Implement the `supportsFindingDuplicates` method. Return `true` if the external system is capable of finding and returning duplicate contacts. If this method returns `false`, then a call to the `findDuplicates` method might throw an exception, but in any event cannot be relied upon to return legitimate results.

Finding duplicate contacts

PolicyCenter 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 PolicyCenter user to determine if the duplicate is valid.

The list of properties returned by ContactManager in its default configuration is in the `ABCContactAPIFindDuplicatesResult` class in the package `gw.webservice.ab.ab900.abcontactapi`.

See also

- *Guidewire 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 PolicyCenter sending new contacts to the external contact management system, implement the `addContact` 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 might be side effects on the local contact in PolicyCenter, typically to store external identifiers.

Methods for adding a contact to an external system

The `addContact` method has two signatures.

```
addContact(Contact contact, String payload, String transactionId)
addContact(Contact contact, String transactionId)
```

The `payload` parameter uniquely describes the contact for the external system in whatever format the external system accepts, such as XML specified by a published XSD. For a typical integration, your Event Fired rules generate an XML serialization of your contact. The `payload` parameter represents the `Message.Payload` property of the message.

The simpler method signature takes a `Contact` entity and a transaction ID.

The transaction ID parameter uniquely identifies the request. Your external system can use this transaction ID to track duplicate requests from PolicyCenter.

Capturing the contact ID from the external system

The `addContact` method returns nothing to PolicyCenter. However, your implementation of the method must capture the ID for the new contact from the external system. Use the captured value to set the address book UID on the local PolicyCenter contact before your `addContact` 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 PolicyCenter user updates a contact's information, PolicyCenter 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.

Overwriting the local contact with latest values

The `updateContact` method of the contact system plugin updates the external system from a local contact. In contrast, the `overwriteContactWithLatestValues` method overwrites a local PolicyCenter contact with the latest values from the contact in the external system.

The `overwriteContactWithLatestValues` method takes two parameters.

- The `Contact` object to update
- The address book UID of the contact in the external system

A typical contact system plugin performs the following steps.

- Validate the auto synchronization state of the local contact with the external system
- Retrieve the contact from the external system
- Overwrite the local contact from fields in the external system

Configuring how PolicyCenter handles contacts

You can customize some ways that PolicyCenter handles contacts by implementing the `IContactConfigPlugin` plugin interface. There is a built-in implementation that performs the default behavior. You can either customize this implementation (`gw.plugin.contact.impl.ContactConfigPlugin`) or write your own version of this plugin.

Configuring available account contact role types

You can customize how PolicyCenter determines the `AccountContactRole` types available for use in the system. A Gosu implementation of this plugin implements this as a `property get` function for the `AvailableAccountContactRoleTypes` property. A Java implementation of this plugin would implement a `getAvailableAccountContactRoleTypes` method that takes no arguments.

The default implementation gets the list of `AccountContactRole` typecodes and returns all roles that are available. In this context, available means that all the following are true.

- The subtype key is non-null.
- The subtype key is not retired and not abstract.
- The role subtype is enabled.

Refer to the `gw.plugin.contact.impl.ContactConfigPlugin` implementation for an example of an efficient implementation of this method.

Configuring mapping of contact type to account contact role

You can configure how PolicyCenter maps contact types to account contact roles. PolicyCenter calls this plugin's `canBeRole` method to determine whether a contact type can play a specific account contact role.

The method's first parameter is a `ContactType` typecode, such as `Person` or `Company`. The second parameter is an `AccountContactRole` typecode. Your plugin method must return `true` if the contact type can play the account contact role. Otherwise, return `false`.

Configuring account contact role type display name

You can configure the display name of an account contact role. Implement the plugin method `getAccountContactRoleTypeDisplayName`. It takes an `AccountContactRole` typekey and returns a `String` encoding of the localized name of the account contact role.

Configuring account contact role type for a policy contact role

You can customize how PolicyCenter gets an account contact role for a policy contact role. PolicyCenter calls this plugin's `getAccountContactRoleTypeFor` method to perform this lookup. It takes a `PolicyContactRole` typekey and returns an `AccountContactRole` typekey that is associated with the policy contact role.

Configuring allowed contact types for policy contact role type

You can customize how PolicyCenter maps allowed contact types for policy contact role types to contact types. PolicyCenter calls this plugin's `getAllowedContactTypesForPolicyContactRoleType` method to perform this mapping. This method takes a `PolicyContactRole` typecode and returns an array of contact types (the return type is `ContactType[]`). See the built-in implementation for the default implementation.

Configuring whether to treat an account contact type as available

You can customize how PolicyCenter considers an account contact type available on an account contact role. PolicyCenter calls this plugin's `isAccountContactTypeAvailable` method to perform this mapping. This method takes a `AccountContactRole` typecode as a method parameter. Return `true` if the account contact role is available.

Synchronizing contacts with accounts

You can customize how PolicyCenter synchronizes contacts with accounts by implementing the `IAccountSyncablePlugin` interface. This set of entities include entities that implement the interface `AccountSyncable`.

Account syncable entity types include the following entities.

- `PolicyAddlInsured`
- `PolicyAddlInterest`
- `PolicyAddress`
- `PolicyBillingContact`
- `PolicyContactRole`
- `PolicyDriver`
- `PolicyLaborClient`
- `PolicyLaborContractor`
- `PolicyLocation`
- `PolicyNamedInsured`
- `PolicyOwnerOfficer`
- `WCLaborContact`
- `WCPolicyContactRole`

PolicyCenter calls this plugin interface to handle the account synchronization.

The only method is `refreshAccountInformation`, which must refresh any necessary account information to ensure the account syncable will use the most current data when calling the other methods. It takes an entity instance of a type that implements the main `AccountSyncable` interface.

The default implementation of this plugin simply calls the `refreshAccountInformation` method on the `AccountSyncable`.

In addition to the `refreshAccountInformation` method, the `AccountSyncable` interface (which account syncable entities implement) has many methods. The easiest way of seeing how to implement them in Gosu is to look at the Gosu class `gw.api.domain.account.AbstractAccountSyncableImpl`. It is an abstract class that is the superclass for delegate adapters for account syncable objects.

Account contact plugin

You can customize how PolicyCenter copies custom properties from an account contact to another account contact. Implement the account contact handling plugin interface (`IAccountContactPlugin`) to customize this logic. There is a built-in implementation `AccountContactPlugin.gs` with the default behavior. You can create your own implementation of this plugin to customize the behavior, which is important to accommodate your data model extensions to the `PolicyContact` or `AccountContact` entities.

Do not confuse the `IAccountContactPlugin` plugin with the `IContactSystemPlug`, `IContactConfig` plugin, or the `IContactSearchAdapter` plugin.

Suppose you have data model extension properties or arrays on an `AccountContact` or one of its subtypes. You probably need to copy them if an account contact entity is cloned to another account contact. Implement the plugin method `cloneExtensions` to copy these properties data model extension properties. The default implementation does nothing.

It is critical that you not make any changes to the related `PolicyPeriod` entities, `Account` entities, or any other entity in this method. PolicyCenter calls this method while binding a `PolicyPeriod` entity. If you modify any entity other than the new account contact, the policy could become out of sync and create serious problems later on.

This method must not make changes to a related `PolicyPeriod`, `Account`, or other entity in this method or serious problems might occur.

The method signature is shown below.

```
override function cloneExtensions(oldContact : AccountContact, newContact : AccountContact)
```

If you have multiple `AccountContact` subtypes, check the runtime subtype by checking `contact.Subtype` if your behavior depends on the subtype.

Account contact role plugin

PolicyCenter calls the `IAccountContactRolePlugin` plugin to perform some tasks related to `AccountContactRole` entity instances.

After cloning one account contact role to another, you can customize the how PolicyCenter copies over data model extensions from an account contact role to another. Implement the `cloneExtensions` method, which takes the old entity followed by the new one. Copy over any data that you want in the new account contact role.

To support copying pending updates for account contact roles, implement the `copyPendingUpdatesForAccountContactRole` method. It takes the source contact role, followed by the newly cloned contact role.

Contact web service APIs

The web service `ContactAPI` provides external systems a way to interact with contacts in PolicyCenter. 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 PolicyCenter requires a transaction ID, which you must set in the SOAP request header for the call.

Use of XML (XSD-based) types is an important part of the `ContactAPI` web service. For nearly all these XSD-based types, you can find an XSD or Guidewire XML (GX) Model in Studio to represent the object.

For example, the `addContact` method uses the following argument.

```
function addContact(externalContact :  
    gw.webservice.pc.pc1000.gxmodel.contactmodel.types.complex.Contact) : String
```

In this fully-qualified path, the part of the path before `types` indicates the path of the XML model in Studio. In Studio, the path is:

```
configuration/gsrc/gw/webservice/pc/pc1000/gxmodel/ContactModel.gx
```

You can edit `ContactModel.gx` to update the XML definition.

Deleting a contact

You can delete a contact in PolicyCenter from an external system by using methods in `ContactAPI`. There are two variants that take different parameters and return different values. Both methods cause PolicyCenter to remove the contact across all accounts.

Determining whether a contact can be deleted

You can check whether contacts can be deleted from an external system by using the `isContactDeletable` method. There are two variants of this method, which support different styles of IDs.

- If you want to specify the objects by public IDs, use the `isContactDeletableByPublicID` method.
- If you want to specify the objects by address book UIDs, use the `isContactDeletable` method.

The return value is `true` if the contact can be deleted or `false` if the contact cannot be deleted.

PolicyCenter prevents you from deleting contacts that are in use. A contact is in use if it has any of the following qualities.

- The contact is used by an account.
- The contact is used by a `PolicyPeriod`.
- The contact has any account only roles.
- The contact is referenced from a user (a subobject of `User`).
- The contact is a participant in a reinsurance agreement.
- The contact is a broker in a reinsurance agreement.
- The contact is a local-only contact, which means that auto sync is disabled for the contact.

If you try to delete a contact that is in use, the APIs throw an exception.

If the account is in use by an account, the exception message mentions which account uses it. If there are multiple accounts that use it, only one account is mentioned in the message.

The following Java example demonstrates this API:

```
Boolean isDeleteableContact = contactAPI.isContactDeletableByPublicID("pc:12345");
```

Deleting a contact by specifying the public ID

The `removeContactByPublicID` method takes a public ID for the contact (a `String`). This method returns the public ID that you passed in, with no change.

The following Java example demonstrates the API.

```
contactAPI.removeContactByPublicID("pc:1234");
```

The method finds all account contacts (`AccountContact` objects) in the database and tries to delete the contact.

Deleting a contact by specifying AddressBookUID

The `removeContact` method takes the following argument.

- An address book UID (a `String`). This ID is unique and is separate from the public ID. The address book UID corresponds to the external contact management system's native internal ID for this contact.

The `removeContact` method has no return value.

In addition to the limitations of deletion mentioned earlier, this method also requires that the contact is auto synced to an external address book.

The following Java example demonstrates the API.

```
contactAPI.removeContact("12345");
```

Adding a contact

To add a contact to PolicyCenter, call the `addContact` method.

To add a new contact you must populate and provide as an argument a contact in a specific XSD-defined XML object that represents the contact. The fully qualified type name is `gw.webservice.pc.pc1000.gxmodel.contactmodel.types.complex.Contact`. A Guidewire GX model defines this XML type.

The method returns the public ID of the new contact.

The method updates the contact only if auto sync is enabled for that contact.

The following Java example demonstrates the API.

```
gw.webservice.pc.pc1000.gxmodel.contactmodel.types.complex.Contact contactXML;  
  
// here you would populate fields in the contactXML object, including the PublicID property and  
// any other required fields. See the Data Dictionary for details  
  
contactAPI.addContact(contactXML, "my-transaction-ID-12345");
```

See also

- “GX models” on page 605

Updating a contact

You can update a contact from an external system by using the `updateContact` method. The method has no return value.

To add a new contact you must populate and provide a contact as the first argument. Specify the contact in a XSD-defined XML object. The type’s fully qualified name is shown below.

```
gw.webservice.contactapi.beanmodel.XmlBackedInstance
```

Unlike some other XSD-based types used in `ContactAPI`, this type does not contain a corresponding Guidewire GX model file in PolicyCenter.

The `XmlBackedInstance` is a type defined in the `BeanModel.xsd` file. You can find the XSD file in Studio. The `XmlBackedInstance` type in the XSD is an XML container for a series of pairs of property name and property value, both as `String` values.

To determine which contact to update, PolicyCenter uses that object’s address book UID, which is in the `LinkID` property in the `XmlBackedInstance` object. If the contact cannot be found, PolicyCenter throws an exception. Finally, PolicyCenter uses the other properties in the contact XML object to update the contact entity instance.

The method updates the contact only if automatic synchronization (`autosync`) is enabled for that contact.

The following Java example demonstrates the APIs.

```
gw.webservice.contactapi.beanmodel.XmlBackedInstance contactXML;  
  
// here you would populate fields in the contactXML object, especially the LinkID property,  
// which contains the Address Book UID for this contact  
  
contactAPI.updateContact(contactXML);
```

Merging contact addresses

You can merge two contact addresses from an external system. You need to know the IDs of both addresses, and you must decide which one to survive after the merge. There are two variants of this method, which support different styles of IDs.

- If you want to specify the objects by public IDs, use the `mergeContactAddressesByPublicID` method. The return value is the public ID of the surviving address.
- If you want to specify the objects by address book UIDs, use the `mergeContactAddressesByABUID` method. The return value is the address book UID of the surviving address.

Pass the following parameters in the listed order.

- The ID of the contact that contains both addresses.
- The ID of the address to merge and then survive. This is called the surviving address.
- The ID of the address to merge (into the surviving contact) and then destroy. This is called the merged address.

After merging, the following results exist.

- Both **Address** entities are on the **Contact** in the database after merging.
- Foreign key references to the merged address now reference the surviving address.
- PolicyCenter retires (deletes) the old **Address** entity.
- PolicyCenter removes the entry in the contact's **ContactAddress** table for the retired entity.

If any of the contacts cannot be found, the API throws an exception.

The following Java example demonstrates the APIs.

```
contactAPI.mergeContactAddressesByPublicID("pc:1234", "pc:5550", "pc:5551");
contactAPI.mergeContactAddressesByABUID("12345", "55550", "55551");
```

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.

There are two variants of this method, which support different styles of IDs.

- If you want to specify the objects by public IDs, use the `mergeContactsByPublicID` method.
- If you want to specify the objects by **AddressBookUID**, use the `mergeContacts` method.

Both methods return nothing.

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.
3. A transaction ID (a `String`) that uniquely identifies the request from an external system. PolicyCenter performs no built-in check with this parameter in the current release. You can modify PolicyCenter to use this transaction ID to detect, log, and handle duplicate requests. A request is a duplicate if the transaction ID matches a previous request.

Merging contacts has the following results.

- Non-duplicate entities on array properties on a contact are merged onto the kept contact. This includes **Addresses**, **RelatedContacts**, **CategoryScores**, **OfficialIDs**, **Tags**. Duplicate entries are discarded.
- Fields on the deleted contact are not preserved.
- Account contact (**AccountContact**) objects that reference the deleted contact now reference the kept contact. If both exist on the same account, the kept contact's **AccountContact** property value is used.
- Account contact roles (**AccountContactRole[]**) on a merged **AccountContact** move to the kept contact's account contact. If there are duplicate roles, the kept contact's roles are preserved.
- PolicyCenter refreshes the kept contact from the external Contact Management System. PolicyCenter calls the contact system plugin (`IContactSystemPlugin`) method called `retrieveContact(String, Bundle)`.
- If the deleted contact was an account holder (`Account.AccountHolder`), PolicyCenter makes the kept contact active and gives it the `AccountHolder` role on that account.
- Policy contact roles (**PolicyContactRole[]**) that reference a merged **AccountContactRole** change to reference the kept contact's **AccountContactRole**. Duplicate **PolicyContactRole** objects remain on the policy, but raise a validation error on Quote or Bind actions.
- The deleted contact and any duplicate subobject is retired (deleted). This includes the following: **AccountContact** and **AccountContactRole**.

The following Java example demonstrates the API method.

```
contactAPI.mergeContacts("pc:uid:1234", "pc:uid:5550", "my-transaction-ID-12345");
```

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.

Activating and deactivating contacts

Deactivating a contact marks a contact as used historically on a policy or account but unused for future policies. Deactivation prevents PolicyCenter from adding that contact to policies. If an inactive contact is already on a policy, the next job that starts on the policy must remove the contact.

You typically use the deactivation feature in integrations only if your external contact management system implements the concept of active and inactive contacts.

To activate or deactivate a contact on all accounts, call the `activateContactOnAllAccounts` method.

The `activateContactOnAllAccounts` method takes two parameters.

- A contact public ID (a `String`)
- A Boolean value that specifies whether to activate or deactivate the contact. If you pass `true`, PolicyCenter activates the contact. If you pass `false`, PolicyCenter deactivates the contact.

The method returns an integer that indicates the number of contacts that the API updated.

The API throws an exception if either of the following occurs.

- The account or contact cannot be found.
- You try to deactivate an account holder.

The following Java example demonstrates the API.

```
Integer totalUpdated;
totalUpdated = contactAPI.activateContactOnAllAccounts("pc:1234", true /* activate */);
```

Getting associated policy transactions for a contact

You can get associated policy transactions for a contact from an external system with method on `ContactAPI`. You can merge two contact addresses from an external system. You need to know the IDs of both addresses, and decide which one to survive after the merge. There are two variants of the method which support different styles of IDs.

- If you want to specify the contact by public ID, use the `getAssociatedWorkOrdersByPublicID` method.
- If you want to specify the contact by address book UID, use the `getAssociatedWorkOrders` method.

Both methods return an array of policy transaction (job) objects. The job objects are XSD-defined XML objects with the type fully qualified name.

```
gw.webservice.pc.pc1000.gxmodel.jobmodel.types.complex.Job
```

A Guidewire GX model defines this XML type.

Pass the following parameters in the listed order.

- The ID of the contact.
- The work status code of the policy period to search. This is a `String` representation of the policy period status typecode from the `PolicyPeriodStatus` typelist.

The following Java example demonstrates the APIs.

```
Job[] workOrderList;
workOrderList = contactAPI.getAssociatedWorkOrdersByPublicID("pc:1234", "BOUND");
```

Getting associated policies for a contact

Get associated policies for a contact with the `getAssociatedPolicies` method. There are two variants of the method which support different styles of IDs.

- If you want to specify the contact by public ID, use the `getAssociatedPoliciesByPublicID` method.
- If you want to specify the contact by address book UID, use the `getAssociatedPolicies` method.

Both versions return an array of objects that each summarizes a policy period. The policy period objects are XSD-defined XML objects with the type fully qualified name.

```
gw.webservice.pc.pc1000.gxmodel.policyperiodmodel.types.complex.PolicyPeriod
```

A Guidewire GX model defines this XML type.

The only parameter is the ID of the contact.

The following Java example demonstrates the APIs.

```
gw.webservice.pc.pc1000.gxmodel.policyperiodmodel.types.complex.PolicyPeriod[] policyList;
policyList = contactAPI.getAssociatedPoliciesByPublicID("pc:1234");
```

Getting associated accounts for a contact

Get associated accounts for a contact with the `getAssociatedAccounts` method. There are two variants of the method which support different styles of IDs.

- If you want to specify the contact by public ID, use the `getAssociatedAccountsByPublicID` method.
- If you want to specify the contact by address book UID, use the `getAssociatedAccounts` method.

Both versions return an array account summary objects. The account objects are XSD-defined XML objects with the type fully qualified name.

```
gw.webservice.pc.pc1000.gxmodel.accountmodel.types.complex.Account
```

A Guidewire GX model defines this XML type.

The only parameter is the ID of the contact.

The following Java example demonstrates the APIs.

```
gw.webservice.pc.pc1000.gxmodel.accountmodel.types.complex.Account[] accountList;  
accountList = contactAPI.getAssociatedAccountsByPublicID("pc:1234");
```

Address APIs

The AddressAPI lets external systems interact with addresses in PolicyCenter. This web service is WS-I compliant. There are several ways to update addresses, with options such as linking and unlinking the address with other addresses.

Refer to the implementation class `gw.webservice.pc.VERSION.AddressAPI.gs` for implementation details of the methods in this web service.

Updating an address

You can update an address using methods on the AddressAPI web service.

There are several methods that update addresses, compared in the following table.

Method name	Purpose
updateAddressOnly	Updates the address but not its linked addresses
updateAddressAndLinkedAddresses	Updates the address and its linked addresses
updateAddressAndUnlink	Updates the address and unlink its linked addresses
updateAddress	In the base configuration, this method has the same behavior as the updateAddressAndLinkedAddresses method. This method is designed to represent the default behavior for each insurer. Modify this method's implementation as appropriate. Note that the updateAddress method calls the AddressService class. The AddressService.updateAddress(...) method determines which update algorithm to use by default. You can modify that method to change the behavior.

For all versions of the method, pass the following parameters in the listed order.

- The address data encapsulated in an object of type `gw.webservice.pc.contact.AddressData`. This is a Gosu class. You can read the source code of the file in Studio.
- The public ID (a `String`) of the address object.

The following Java example demonstrates the APIs.

```
gw.webservice.pc.contact.AddressData updateAddress;  
  
// here you would populate fields in the object in the updateAddress variable  
// for example the PublicID property and  
// any other required fields. See the Data Dictionary for details  
  
addressAPI.updateAddress(updateAddress, "pc:1234");
```

Part 9

Importing policy data

Importing from database staging tables

PolicyCenter includes zone mapping functionality used by the assign by location, address auto-fill, and regional holidays features. PolicyCenter supports high-volume bulk import of zone data by using database staging tables. This topic describes the use of database staging tables to import zone data.

PolicyCenter supports database staging table import of zone data only. PolicyCenter does not support staging table import of policies, accounts, or other business or administrative data. Use the standard Guidewire APIs to import non-zone data. Implement segmentation strategies to improve loading performance if necessary.

PolicyCenter uses zone data for the following features.

- Assignment by location
- Address auto-fill
- Setting regional holidays

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 PolicyCenter regularly.

A staging table provides an intermediate data location between a comma-separated values (CSV) file and the operational zone table in the database. You set up the CSV file to contain the data columns that you need. Guidewire provides tools to load the CSV data into the staging table. You use other Guidewire tools to check the integrity and consistency of the data in the staging table. Correcting errors in the staging table data reduces operational down time caused by rolling back failed data imports. You complete the import process by loading the data into the operational tables. This final step uses bulk SQL Insert and Select statements that operate on the entire zone table to ensure high performance.

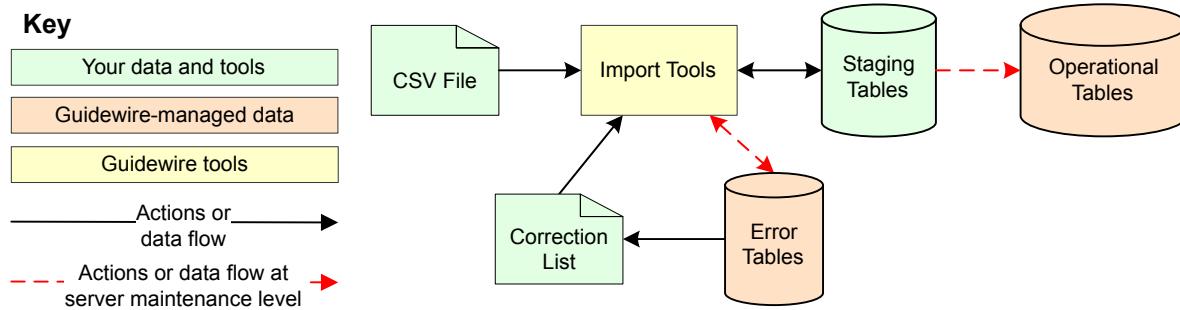
Importing zone data

In PolicyCenter, the database staging tables and the database import tools load zone data only. The sources for zone data are CSV files for each country or region that you need to support. PolicyCenter 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.

Zone Import Typical Flow



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 PolicyCenter.

- 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 PolicyCenter web services or command prompt table import tools. Set the server to the multiuser run level.

See also

- *System 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

PolicyCenter 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, PolicyCenter 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

PolicyCenter 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*

Database import tables and columns

Importing data into the PolicyCenter 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. PolicyCenter displays this information in info pages in the server tools user interface.

Never interact with any component of the PolicyCenter database directly, other than staging tables. Instead, use PolicyCenter 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

- *System Administration Guide*

Staging tables

Staging tables are database tables that replicate the loadable columns of specific operational tables in the PolicyCenter database. The operational zone table `pc_zone` has a corresponding staging table `pcst_zone`. You prepare zone data for bulk import into the PolicyCenter operational table `pc_zone` by first loading the data into the `pcst_zone` staging table.

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, a Zone record for a postal code must load into the pcst_zone staging table at the same time as a Zone record for its city. The zone import tool creates an identifier (LUWID) for each LUW that it defines.

After staging table data loads successfully into operational tables, PolicyCenter 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, you must correct the data in the CSV file and reload the staging table data.

See also

- *System Administration Guide*

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 PolicyCenter at

Server Tools→**Info Pages**→**Load History**.

See also

- *System 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. PolicyCenter 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 PolicyCenter through staging table import. For example, a row that a user created by using the PolicyCenter 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

- *System Administration Guide*

Load user IDs

Running a staging table import populates user ID properties on imported entities. PolicyCenter 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 PolicyCenter 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 PolicyCenter.

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. PolicyCenter 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

PolicyCenter 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 `PolicyCenter/admin/bin` and view the built-in help.

```
table_import -help
```

PolicyCenter 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 `PolicyCenter/admin/bin` and view the built-in help.

```
zone_import -help
```

See also

- “Table import web service” on page 107
- “Zone data import web service” on page 111
- *System 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 PolicyCenter 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 104
- *System Administration Guide*

Database consistency checks

Database consistency checks ensure that the data in the PolicyCenter 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.

Optionally, after loading the staging table data into the operational tables on the live PolicyCenter database, run consistency checks again.

Example commands are shown below.

- API method: `SystemToolsAPI.submitDBCCBatchJob()`
- Command prompt tool: `system_tools -passwordpassword -checkdbconsistency`

See also

- *System Administration Guide*

Integrity checks

The PolicyCenter 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 PolicyCenter 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, PolicyCenter runs integrity checks. If the checks fail, PolicyCenter 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, PolicyCenter must rerun integrity checks before importing your data.

- 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 PolicyCenter user interface. Importing a `null` value in this property passes integrity checks. However, if a user uses the

PolicyCenter interface to edit an object containing the property, PolicyCenter requires a non-null value before saving because of data model validation.

- Integrity checks are different from validation rule sets and the validation plugin.

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
```

See also

- *System Administration Guide*

Types of integrity checks

The following partial list of data integrity checks shows the types of items that PolicyCenter enforces.

- No duplicate PublicID strings within the staging tables or in the corresponding operational tables
- No unmatched foreign keys
- No missing, required foreign keys
- 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

Database performance considerations

- 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 -updatedatabasestatistics -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 111
- *System Administration Guide*

Clearing errors from staging tables

The data rows in staging tables must not contain errors that cause integrity checks to fail. You must correct the data in your zone data CSV files. Then, clear the staging tables and reload the data to the staging tables.

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
```

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, PolicyCenter 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.

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 PolicyCenter 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, PolicyCenter 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.

PolicyCenter 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 PolicyCenter 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, PolicyCenter ignores this parameter and option.

Estimate the database statistics only on Oracle databases that contain few or no policy records. Estimating the statistics on databases that have more data significantly reduces database performance.

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.

To import zone data to PolicyCenter, you typically perform the steps described in the subsequent sections.

Prepare the data and the PolicyCenter 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. Create zone data files. Create zone data files in comma separated value (CSV) format.
3. Set the server to the maintenance run level. Set each PolicyCenter server to the maintenance run level by using the `system_tools` command prompt tool or the web service `SystemToolsAPI.setRunLevel` method.
4. Back up the operational tables in the database. Back up operational tables before import.
5. Clear the staging tables, the error tables, and the exclusion tables. Use web service APIs or command-line tools.
6. Run PolicyCenter 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. Run the zone import tool. Run the command prompt tool `zone_import` or use the `IZoneImportAPI` web service to import your CSV data.
2. Set the server to the maintenance run level. Set each PolicyCenter server to the maintenance run level by using the `system_tools` command prompt tool or the web service `SystemToolsAPI.setRunLevel` method.
3. 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.

4. Set the server to the multiuser run level. Set each PolicyCenter server to the multiuser run level by using the `system_tools` command prompt tool or the web service `SystemToolsAPI.setRunLevel` method.
5. Check load errors from the Load History page. This page is available at **Server Tools**→**Info Pages**→**Load History**.
6. Correct errors in the zone data source files. Edit the zone data CSV file to correct the data.
7. 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.

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 PolicyCenter.

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. Set the server to the maintenance run level. Set each PolicyCenter server to the maintenance run level by using the `system_tools` command prompt tool or the web service `SystemToolsAPI.setRunLevel` method.
3. Load staging table zone data into operational zone tables. To load only zone table data, use a method that begins with `integrityCheckZoneStagingTableContentsAndLoadZoneSourceTables` on the `TableImportAPI` web service API. Alternatively, use the `table_import` command prompt tool with the `-integritycheckandload` option. PolicyCenter inserts data rows into the operational zone 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 -zonedataonly
```

 - For a non-Oracle database or an Oracle database that already contains many records, use the following command.

```
table_import -password password -integritycheckandload -zonedataonly
```
4. Set the server to the multiuser run level. Set each PolicyCenter 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 PolicyCenter 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. If necessary, import additional records into the staging tables.

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 PolicyCenter application.
- PolicyCenter 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, PolicyCenter provides table-specific update database statistics commands in the `cc_loaddirstatistics` 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.
- During PolicyCenter upgrade, the PolicyCenter 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.

Part 10

Other integration topics

Guidewire InsurancePlatform Integration Views

Integration Views allows you to select and retrieve data from the PolicyCenter database and further compose and transform that data in a custom manner to meet business needs. Moreover, the feature allows you to define a stable, versioned contract for this data processing. In addition, the feature provides a simple, declarative way to map underlying PolicyCenter data into a serialized format that conforms to the contract.

You define the contract explicitly and independently from the underlying PolicyCenter data model. In this way, the contract is decoupled from the data model. Because of this manner of definition, when you change the contract, the change is both explicit and intentional.

Integration Views allows different versions of the contract to coexist. This simultaneous existence allows you to evolve the contract and the systems that rely upon it in a controlled fashion to meet business needs.

At the same time, Guidewire can improve the PolicyCenter data model. By strategically decoupling your data integration process from the Guidewire data modeling process, Integration Views allows 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 PolicyCenter 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.

See also

- “Mapping a data object to an Integration View schema” on page 598
- “Creating a schema for Integration View objects” on page 599
- “Filtering Integration View objects for select attributes” on page 601
- “Handling an Integration View using a REST API” on page 602

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 mappers 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 601

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.

See also

- “Accessing and naming files related to Integration Views in Guidewire Studio” on page 601
- <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 752

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 752
- “JSON data types and formats” on page 764

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 601

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 PolicyCenter 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 PolicyCenter 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 601

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.

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 `AddressModel` 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 PolicyCenter 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.

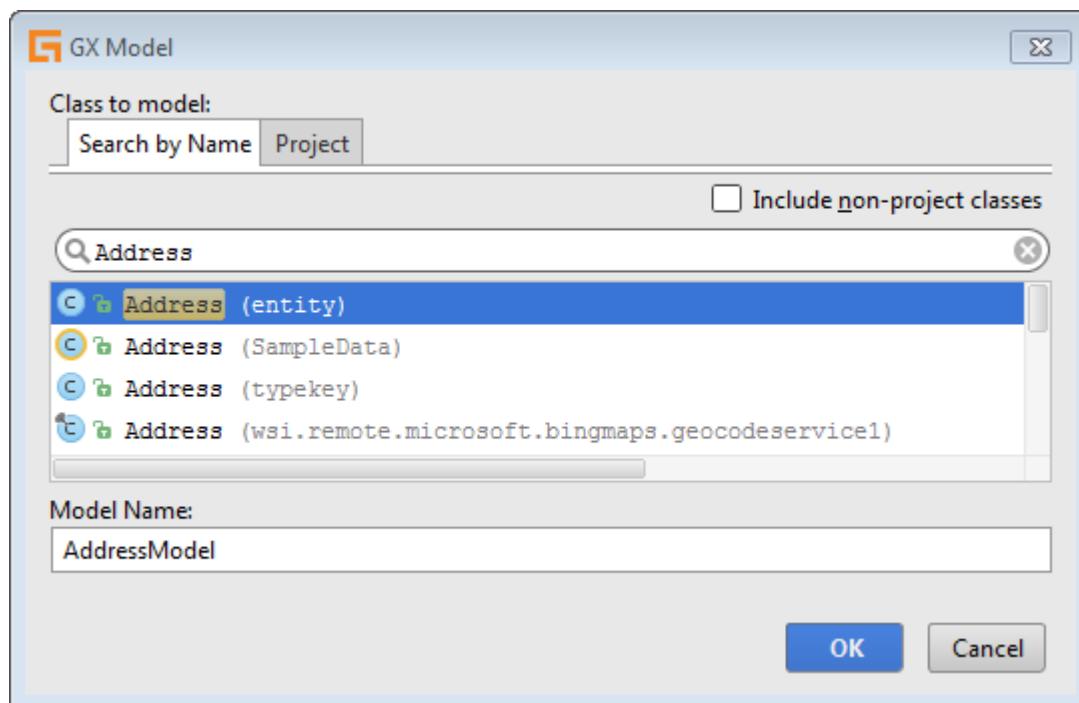
When configuration code subsequently creates an instance of the GX model, the constructor methods accept an argument of an instance of the data type associated with the model. The properties of the argument that are referenced by the GX model are stored in the new instance. They can be subsequently converted to XML that conforms to the model's XSD schema. Continuing with the `Address` and `AddressModel` example, an instance of the `AddressModel` 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.

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.

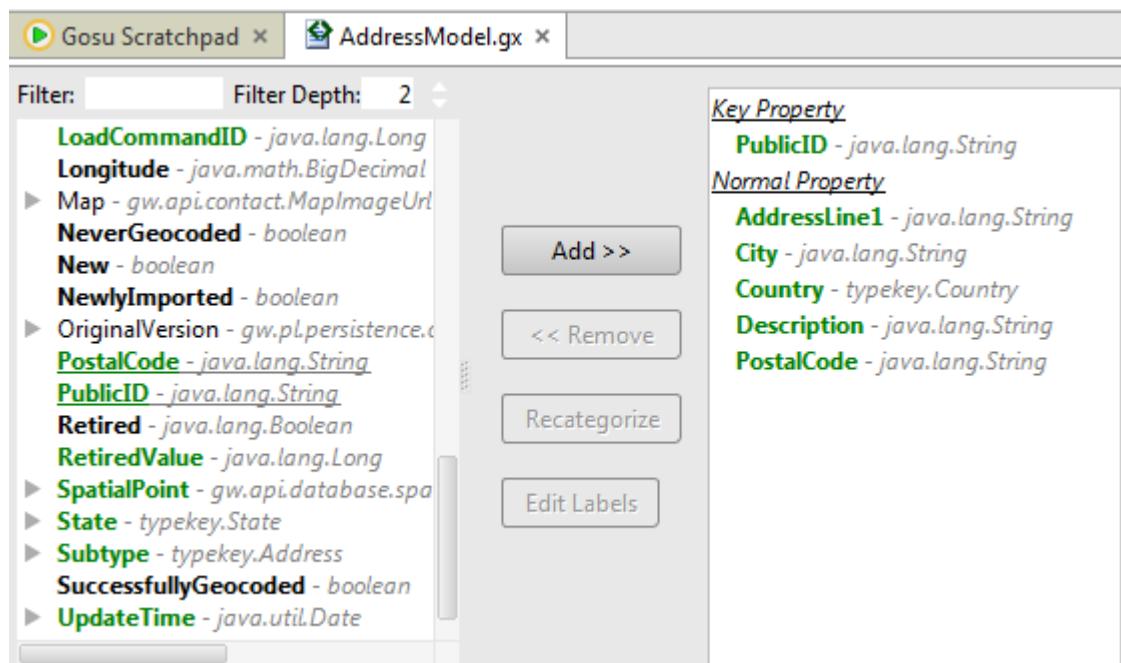
4. Select either the **Search by Name** or **Project** tab to list the available classes and data types to map.
5. In the Search field, enter the name of the type to export in XML format. For example, to map and export an Address entity, type Address. The dialog box lists the types that match any partial 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 `Model`. For example, if the selected class is `Address`, the model name is `AddressModel`.

The GX model name defines the package hierarchy for the model, so choose its name carefully. For example, if an XML model called `AddressModel` is created in the `mycompany` package, the fully qualified type for the `Address` entity in the GX model is `mycompany.addressmodel.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 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.

Result

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.

Next steps

For more information about key properties and normal properties in the context of the incremental mode, see "The Incremental Option" under "GXOptions" on page 609.

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 `Policy` 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 PolicyCenter application is running, the application publishes the GX model's XSD schema. Other applications can import the schema from PolicyCenter and use it to validate related XML files.

PolicyCenter publishes the XSD schema at the following URL.

```
http://HOSTNAME:PORT/pc/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, PolicyCenter 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` set to `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 label1001 = com.mycompany.addressmodel.Address.create(myAddress, {}, {"myLabel1001"})
var label1002 = com.mycompany.addressmodel.Address.create(myAddress, {}, {"myLabel1001", "myLabel1002"})
var label1003 = com.mycompany.addressmodel.Address.create(myAddress, {labelSysA}, {"myLabel1001"})
var label1004 = 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 `GxModel11` defines a property `Property1_1` with label `SystemA`. Assume also that a GX model `GxModel12` defines a property `Property2_1` with a label having the same name, `SystemA`. The respective static property references for the `SystemA` label—`GxModel11.Label.SystemA` and `GxModel12.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 `GxModel11` and `Property2_1` on `GxModel12`.

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`. PolicyCenter 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>

Gosu type	GX Model XSD type
gw.xml.xsd.types.XSDDate	xsd:date
gw.xml.xsd.types.XSDDateTime	xsd:dateTime
gw.xml.xsd.types.XSDDuration	xsd:duration
gw.xml.xsd.types.XSDGDay	xsd:gDay
gw.xml.xsd.types.XSDGMonth	xsd:gMonth
gw.xml.xsd.types.XSDGMonthDay	xsd:gMonthDay
gw.xml.xsd.types.XSDGYear	xsd:gYear
gw.xml.xsd.types.XSDGYearMonth	xsd:gYearMonth
gw.xml.xsd.types.XSDTime	xsd:time
int	xsd:int
java.lang.Boolean	xsd:boolean
java.lang.Byte	xsd:byte
java.lang.Double	xsd:double
java.lang.Float	xsd:float
java.lang.Integer	xsd:int
java.lang.Long	xsd:long
java.lang.Short	xsd:short
java.lang.String	xsd:string
java.math.BigDecimal	xsd:decimal
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

Archiving integration

PolicyCenter supports archiving policy terms as serialized streams of data, with one serialized stream per `PolicyPeriod` entity in the term. Each serialized stream is an XML document. You can choose how to store the XML documents in an external system. Store the data as files, as database binary large objects, or documents in a document storage system.

From the user perspective, PolicyCenter archives or retrieves a *policy term*. A policy term is the *logical unit of archiving*.

PolicyCenter does not store a policy term as a single XML document. Instead, PolicyCenter serializes and stores one XML document for every `PolicyPeriod` graph in the policy term. A `PolicyPeriod` graph means one `PolicyPeriod` object and its subobjects. The `PolicyPeriod` graph is the *physical unit of archiving*. PolicyCenter archives each `PolicyPeriod` graph in the term sequentially in an internally-defined order. PolicyCenter stops archiving that term if there are any errors with any `PolicyPeriod`.

After archiving, PolicyCenter deletes most `PolicyPeriod` subobjects. However, PolicyCenter retains the `PolicyPeriod` entity instance and its associated entity instances of type `EffectiveDatedFields` and `Document`. Additionally, in certain circumstances certain other objects are retained:

- PolicyCenter deletes `Note` objects associated with the job of the archived policy period if and only if it has no associated activity.
- PolicyCenter deletes `UWIssueHistory` objects if and only if it is an auto-approved issue
- PolicyCenter deletes `FormTextData` objects only with the last remaining `PolicyPeriod` to archive in that term. In other words, it is always deleted, but in the last database transaction along with the last remaining archived `PolicyPeriod` in the term.

PolicyCenter documentation refers to a `PolicyPeriod` object as the *root info object* for that archived data. The name root info object refers to the `RootInfo` delegate, which the `PolicyPeriod` entity type implements. Some archiving APIs use the `RootInfo` object as a method argument or return type. For PolicyCenter, when you see `RootInfo`, this refers to a `PolicyPeriod` instance.

After archiving, PolicyCenter makes the `PolicyPeriod` data effectively immutable while that period is archived. However, within one policy term, at any given instant, it is possible that one or more periods are archived but not all. From the user interface standpoint, PolicyCenter considers a given policy term archived if one or more of the periods in the term is archived, even if not all are archived.

If the plugin throws exceptions, the application sets the `PolicyPeriod.ExcludedFromArchive` Boolean property to `true` and sets the `PolicyPeriod.ExcludeReason` to a description `String`. Check your logs for errors. If you have a coding problem that sets this property, use the **Archive Info** screen to reset the `PolicyPeriod.ExcludedFromArchive` flag to `false`.

There are two archiving-related plugin implementations in PolicyCenter. Both demonstration implementations are in the `gw.plugin.archive.impl` package. The following table summarizes the role of each interface:

Plugin interface	Description	Implementation class
IArchiveSource	Provides methods to store and retrieve a serialized XML document that represents one instance of the <code>PolicyPeriod</code> archive domain graph.	PCArchiveSourcePlugin
IPCArchivingPlugin	Provides a single method to use to determine when to archive a policy term.	PCArchivingPlugin

See also

- [Application Guide](#)
- “Archive source plugin” on page 622
- “Archiving eligibility plugin” on page 629

Overview of archiving integration

Basic integration flow for archiving storage

The following sequence describes the high-level integration flow for archival storage.

1. The PolicyCenter Archive Policy Term work queue runs.
2. PolicyCenter determines the set of `PolicyPeriod` entity instances that need archiving. To determine what `PolicyPeriod` entity instances to skip, PolicyCenter uses the following properties:
 - `PolicyTerm.NextArchiveCheckDate` - Date that determines the earliest archive eligibility. PolicyCenter populates this property from calling the policy eligibility plugin. See “Archiving eligibility plugin” on page 629 for more information.
 - `PolicyPeriod.ExcludedFromArchive` - Boolean flag, which, if set to `true`, prevents PolicyCenter from attempting to archive the policy period. If the value is `true`, PolicyCenter skips the policy period and does not run the archiving rule set for this policy period again until this flag is reset to `false`. For an excluded policy period, the `PolicyPeriod.ExcludedReason` property contains a string value that describes the reason for the exclusion.
3. PolicyCenter performs the following steps iteratively for each unarchived policy period in the term. PolicyCenter chooses the next policy period in an order defined by an internal algorithm. Only one policy period in a term archives at any one time. If a policy period fails archiving, the rest of the policy periods in that term remain unprocessed until an administrator fixes the problem. For problems, the application sets the `PolicyPeriod.ExcludedFromArchive` Boolean property to `true` and sets the `PolicyPeriod.ExcludeReason` to a description String.
4. PolicyCenter encodes each instance of the `PolicyPeriod` graph into an in-memory XML representation.
5. PolicyCenter serializes each XML document.
6. PolicyCenter calls the archive source plugin that you created to store the data.
7. 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.

8. After archiving, PolicyCenter deletes most `PolicyPeriod` subtypes. However, PolicyCenter retains the `PolicyPeriod` entity instance and its associated entity instances of type `EffectiveDatedFields` and `Document`. Additionally, in certain circumstances, PolicyCenter retains certain other entities. See “Archiving integration” on page 617 for details.

Basic integration flow for archiving retrieval

The following sequence describes the high-level integration flow for archiving retrieval.

1. The user identifies a policy term to retrieve.

2. In a work queue, PolicyCenter finds the associated `PolicyPeriod` entities for the policy term. The following steps happen iteratively for each policy period in the term, in an order defined by internal code. Only one policy period is retrieved at one time. If retrieval fails, PolicyCenter creates an activity for all requestors.
3. PolicyCenter tells the archive source plugin to retrieve the XML formatted archival data from the archive backing store.
4. PolicyCenter 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 PolicyCenter, PolicyCenter 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 619

6. PolicyCenter converts the temporary object into a real entity instance, upgrades the data model if necessary, and finally persists the entity instance to the database.
7. PolicyCenter adds activities to notify retrieval requestors of the success of the retrieval operation. There may be more than one requestor for the retrieval of a policy term.

PolicyCenter web service for archiving

To manipulate archiving from systems external to Guidewire PolicyCenter, call the `IArchiveAPI` web service. Use this web service to do the following, for example:

- Check if a policy term is archived
- Suspend a policy from further archiving

This web service is WS-I compliant. For more information, see “Archiving web services” on page 128.

Check for archiving before accessing policy data

Be careful before you access policy data to determine whether the data is already archived. Before traversing the path from a `PolicyPeriod` entity to its linked entities, you must check whether that policy period is archived. If a policy period is archived, no entities other than the `PolicyPeriod` and its associated effective dated fields entities remains accessible. If your code attempts to traverse the entity in an archived `PolicyPeriod`, PolicyCenter throws an exception. The `PolicyPeriod Archived` property determines if the policy period is archived.

The policy term itself has an `Archived` property that is `true` if at least one policy is archived. From the user perspective, the term is archived if any policy periods are archived, even if not all are archived. Therefore, in many cases it is also appropriate to confirm whether the term of the policy period is archived before proceeding with an action.

See also

- *Configuration Guide*

Upgrading the data model of retrieved data

During a PolicyCenter archive retrieval operation, after the archive source plugin retrieves archival data, PolicyCenter deserializes the data into an in-memory object that represents the entity instance. This in-memory object is not yet a real entity instance. PolicyCenter loads the data into an object called an archived object, represented by the interface `IArchivedEntity`.

PolicyCenter then determines if the archive object is from an data model version that is earlier than the current version. If this is the case, PolicyCenter 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:

- PolicyCenter performs built-in upgrade steps such as adding a column to the object.
- PolicyCenter runs customer upgrade steps such as upgrading the data in the data model extension properties.

Only after all upgrade triggers run does PolicyCenter 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.

See also

- *Customizing the database upgrade* in the *Upgrade Guide*

Error handling during archive

If the archiving process fails in any way, consult both of the following:

- Application logs
- The **Archive Info** screen within the **Server Tools** screen.

To view the **Archive Info** screen, you must have administrative privileges.

See also

- *System Administration Guide*
- *Configuration Guide*

Archiving storage integration detailed flow

PolicyCenter implements archive integration as a work queue. PolicyCenter 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 PolicyCenter, from the work queue perspective, the eligible entity type for archiving is a policy term. However, a term contains potentially multiple **PolicyPeriod** entities and the **PolicyPeriod** is the physical unit of archiving.

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.

Before attempting archiving, PolicyCenter re-checks some criteria to see if it is eligible even if it already checked this while originally queuing the policy term. For example, an out of sequence job on the term now makes it ineligible even if that job did not exist when the term was queued for archive.

4. For each work item, PolicyCenter performs the following steps within a single database transaction:
 - a. The worker calls the **IArchiveSource** plugin method `updateInfoOnStore`. It takes a single argument, which is a **PolicyPeriod** entity instance. In the plugin definition, this parameter is declared as an entity type that implements the **RootInfo** delegate. If you need to add or modify properties on **PolicyPeriod** prior to archiving, set these properties in your `updateInfoOnStore` method.
 - b. The worker determines the set of entities within the domain graph of this entity instance. To accomplish this task, the worker *tags* the archived root entity instance (the **PolicyPeriod**) specified in the work item. Next, worker recursively tags all entities in the domain graph whose parent entity instance 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.

- c. The worker internally generates a structure that represents an XML document for the archived `PolicyPeriod` and its subtypes.
 - d. 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.
 - e. After archiving, PolicyCenter deletes most `PolicyPeriod` subtypes. However, PolicyCenter retains the `PolicyPeriod` entity instance and its associated entity instances of type `EffectiveDatedFields` and `Document`. Additionally, in certain circumstances, PolicyCenter retains certain other entities. See “Archiving integration” on page 617 for details.
 - f. 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 `PolicyPeriod` entity instance to archive.
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, PolicyCenter rolls back all changes in the entire transaction.
 6. As the last step in the archiving process for each work item, the worker calls the archive source plugin method `storeFinally`. PolicyCenter calls this method independent of whether the archive transaction succeeded. For example, if some code threw an exception and the archive transaction never committed, PolicyCenter 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

- “Archive source plugin storage methods” on page 624
- “Archive source plugin utility methods” on page 627
- *Application Guide*
- *System Administration Guide*
- *Configuration 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. PolicyCenter does not check whether the database has a newer version of the record to retrieve from the archive.

Within PolicyCenter, the archive document retrieval process has the following overall flow:

1. The user initiates the retrieval process on an archived policy term.
2. Asynchronously, the retrieve request work queue finds items to retrieve.
3. PolicyCenter calls the `IArchiveSource` plugin method called `retrieve`. PolicyCenter 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: PolicyCenter turns off field validation during the retrieval of archival data from the archive backing store.

4. PolicyCenter creates any data that relates to the retrieve. For example, notes and history events.
5. PolicyCenter performs any upgrade steps defined for the data.

See “Upgrading the data model of retrieved data” on page 619 for more information.

6. PolicyCenter calls `IArchiveSource.updateInfoOnRetrieve`. This method gives you a chance to perform additional operations in the same bundle as the retrieval operation, after PolicyCenter recreates all the entities but before the commit.

In the demonstration plugin implementation, this method sets the `PolicyTerm.NextArchiveCheckDate` property to a date based on the `ArchiveDaysRetrievedBeforeArchive` configuration parameter. This ensures that the application does not immediately attempt to archive the policy period again but instead waits until it is eligible for archive.

7. PolicyCenter commits the bundle.
8. PolicyCenter 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 web services” on page 128
- “Archive source plugin retrieval methods” on page 626
- “Archive source plugin utility methods” on page 627
- *Application Guide*
- *System Administration Guide*

Archive source plugin

If your PolicyCenter 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

PolicyCenter includes a demonstration implementation of the `IArchiveSource` plugin interface.

```
gw.plugin.archive.impl.PCArchiveSourcePlugin
```

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 PolicyCenter 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, PolicyCenter calls `ArchiveSource` method `store` synchronously, waiting for the method to complete or fail.

Retrieving archival data

To retrieve archival data, PolicyCenter 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 PolicyCenter, the archive root info entity is always a `PolicyPeriod` entity. The interface type `RootInfo`, in APIs or in documentation, always refers to `PolicyPeriod`. Your plugin implementation must store enough information in the `PolicyPeriod` entity type and its associated `EffectiveDatedFields` entity to determine how to retrieve any archived document from the archive backing store.

Encrypting archival data

In the base configuration, Guidewire provides the means to encrypt PolicyCenter 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 `DefaultXmlExportIEncryptionId` 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 251.
- For information on how to enable the `IEncryption` plugin, see “Enable your encryption plugin implementation” on page 258.
- For information on how to tag a data field for encryption, see “Setting encrypted properties” on page 251.

Archive source plugin methods and archive transactions

It is important to understand that PolicyCenter 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, PolicyCenter always calls the plugin method “`storeFinally`” on page 625 outside the main database transaction for the storage request. If the storage request fails, PolicyCenter does not commit the data changes to the database. However, any changes you make during the call to `storeFinally` do persist to the database.

WARNING Be extremely careful that you understand potential failure conditions and the relationship between each archive method and associated database transactions.

WARNING 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.

For details for each plugin method, refer to reference information grouped by purpose of the methods:

- “Archive source plugin storage methods” on page 624
- “Archive source plugin retrieval methods” on page 626
- “Archive source plugin utility methods” on page 627

See also

- *Configuration Guide*

Archive source plugin storage methods

PolicyCenter 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 `PolicyPeriod` 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:

- PolicyCenter does not roll back any changes if the archiving operation fails.
- PolicyCenter 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 PolicyCenter, the default plugin implementation of this method records the IDs of the `UWIIssueHistory` entities to delete after the archive operation completes.

`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.

PolicyCenter 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`” on page 625 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 PolicyCenter expects the plugin to throw an error. PolicyCenter 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>)
```

PolicyCenter 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 PolicyCenter 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`” on page 624 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 PolicyCenter 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 PolicyCenter 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 PolicyCenter 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.

PolicyCenter 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`” on page 624.

PolicyCenter calls the `updateInfoOnStore` method inside the archiving transaction. This enables you to make additional updates on `RootInfo` entities.

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`” on page 627 and “`updateInfoonDelete`” on page 629.

WARNING Only modify the `RootInfo` entity (`PolicyPeriod`) in the `updateInfoOnStore` method. It is dangerous and unsupported to modify any other entity instances in the policy period graph in this method.

See also

- *Configuration Guide*

Archive source plugin retrieval methods

PolicyCenter 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 `PolicyPeriod` 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 PolicyCenter 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

PolicyCenter 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 `PolicyPeriod` 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. PolicyCenter 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 PolicyCenter deletes associated entities returned from “`updateInfoonDelete`” on page 629.

`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 PolicyCenter generated the status information.
<code>getDeleteStatus</code>	Returns a typekey from the ArchiveSourceStatus typelist. The typecode is one of the following:
<code>getRetrieveStatus</code>	<ul style="list-style-type: none"> • Available - Archiving service is available.
<code>getStoreStatus</code>	<ul style="list-style-type: none"> • Failure - Last attempt to archive failed. • 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 log the archive issues.

`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` ” on page 628 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 (`PCArchiveSourcePlugin`) 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 `PCArchiveSourcePlugin` plugin is not suitable for production environments.

In addition, this method calls the plugin class “ `refresh` ” on page 628 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 (`PCArchiveSourcePlugin`) 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, PolicyCenter calls the `updateInfoonDelete` method to determine additional entity instances that PolicyCenter must delete if it deletes the entity instance represented by `RootInfo`. PolicyCenter 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`” on page 625 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 PolicyCenter 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 627.

If you create entities in the “`updateInfoOnStore`” on page 625 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.

Archiving eligibility plugin

During the archive process, PolicyCenter calls the `IPCArchivingPlugin` plugin implementation to determine the date on which a policy term is eligible for archiving. In the base PolicyCenter configuration, the `PCArchivingPlugin` class provides the implementation class for this plugin.

The default implementation `PCTermsArchivingPlugin` class provides built-in algorithms for determining the eligibility of a policy term for archiving. For example, if the term has open claims, PolicyCenter does not archive any of the policy periods on that term. However, it is possible for your implementation of the `IPTCArchivingPlugin` to set the date on which to archive a policy term even later than the default logic.

The `PCTermsArchivingPlugin` class contains a single public method, `getArchiveDate`. This method has the following signature.

```
getArchiveDate(policyTerm : PolicyTerm) : Date
```

The default implementation of this method checks for open claims and returns the latest date from the following set of dates:

- Dates of all open claims
- Today's date

See also

- *Configuration Guide*

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

PolicyCenter supports two modes of processing:

- Work queue
- Batch process

Work queue

A work queue operates on a batch of items in parallel. PolicyCenter distributes work queues across all servers in a PolicyCenter 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 policy 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 PolicyCenter 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 PolicyCenter processes

It is possible to schedule many, if not most, of the PolicyCenter processes for execution at periodic intervals. Guidewire recommends that you take the following information into account in setting up a processing schedule.

"Night-time processing" on page 632	Processes business transactions accumulated at the end of specific business periods, such as business days, months, quarters, or years.
"Daytime processing" on page 633	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 PolicyCenter 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 *System 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 PolicyCenter 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 PolicyCenter processes

PolicyCenter 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	<i>System Administration Guide</i>
Work Queue Info	<i>System Administration Guide</i>

Batch processing typecodes

PolicyCenter 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 in 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 638

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 queues 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 override only 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 activities 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 PolicyCenter 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 PolicyCenter 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 PolicyCenter 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

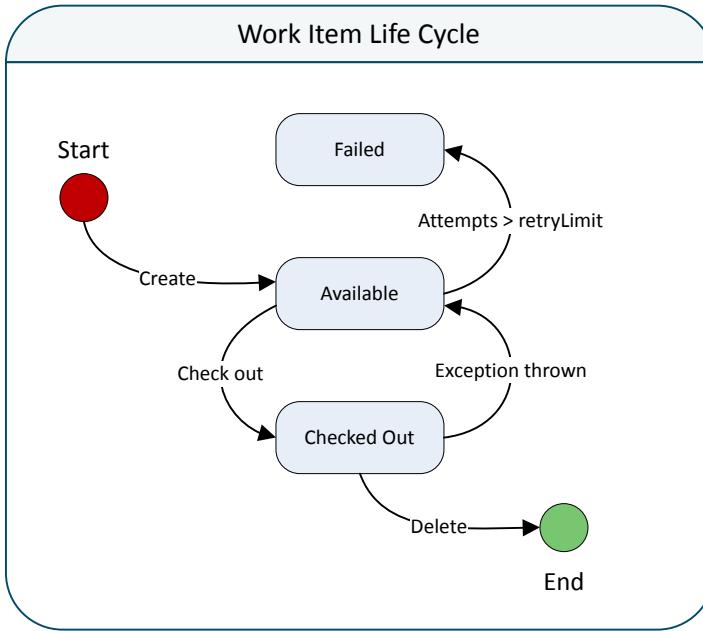
Guidewire does not support creating multiple work queues that use the same work item type, except for the `StandardWorkItem` type. If you attempt to do so using a work item type other than `StandardWorkItem`, PolicyCenter throws the following error:

```
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, PolicyCenter 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 PolicyCenter 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 *System 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, PolicyCenter 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 `softentityreference` is a foreign key for which PolicyCenter 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 PolicyCenter 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 PolicyCenter 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 635

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 634.

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	MyWorkItem
Entity Type	entity
Desc	Custom work item
Table	myworkitem
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:
 - a. 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.
 - b. 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
<code>name</code>	<code>Activity</code>
<code>type</code>	<code>Activity</code>
<code>nullok</code>	<code>false</code>

5. (Optional) Add other fields to your custom entity as meets your business needs.

Next steps

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.

<code>findTargets</code>	Logic for the writer, which selects units of work for a batch and returns an iterator for the result set
<code>processWorkItem</code>	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 643

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. PolicyCenter 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, PolicyCenter 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, PolicyCenter 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 PolicyCenter 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 your 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 PolicyCenter 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. PolicyCenter 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 PolicyCenter calls your custom `processWorkItem` method, but PolicyCenter 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. PolicyCenter 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.

PolicyCenter catches exceptions thrown by code within the scope of your custom `processWorkItem` method. Whenever PolicyCenter 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`, PolicyCenter sets the `Status` property of the work item to `available`. Otherwise, PolicyCenter sets the `Status` property of the work item to `failed`.

Retrying work items that cause exceptions

PolicyCenter 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 PolicyCenter 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.

The following sample code illustrates how to work with the `handleException` method.

```

override function handleException(workItem : W, ex : Throwable,
consecutiveExceptions : int) : boolean {

if (ex typeis MyCustomWorkQueueException) { // Fail immediately for a custom work queue exception
    workItem.fail(DateUtil.currentDate(), ex.getLocalizedMessage())
    return true
} else { // Fail only for other exceptions if attempts exceeds retry count
    return super.handleException(workItem, ex, consecutiveExceptions)
}
}

```

In this method, the arguments represent the following:

- The `W` method argument is a generic variable that represents a `WorkItem` object. This work item can be of type `StandardWorkItem`, or it can be some other custom type, for example, `MyWorkItem` or `MyStandardWorkItem`.
- The `consecutiveExceptions` method argument refers to the number of consecutive exceptions encountered by the worker processing this `workItem` object since the last successfully processed work item. The last successfully processed work item is not necessarily the immediate predecessor of the current work item. It is merely the last work item successfully processed by this worker at any point before the current work item. PolicyCenter

increments the `consecutiveExceptions` counter every time it encounters a problem. PolicyCenter then resets the counter every time that it successfully processes a work item.

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.

PolicyCenter 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`, PolicyCenter 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 PolicyCenter 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 : InsertQueryBuilder, 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:

`opaqueBuilder` The query builder to use.

`allowFailedDuplicates` (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, "Activity not viewed for five days", "See activity" + activity.Subject + ", due on " + activity.TargetDate + ".") // To: // From: // 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 `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 PolicyCenter 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, null, "Activity not viewed for five days", // To: // From: // 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 645

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 633

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 648. If necessary, also review “Custom work queues” on page 633.

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 636 for a discussion as to why Guidewire recommends the use of `softentityreference` instead of `foreignkey` here.

For all other fields, accept that dialog defaults.

- 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> – Activity <code>type</code> – <code>softentityreference</code> <code>nullok</code> – <code>false</code>
<code>implementsEntity</code>	<ul style="list-style-type: none"> <code>name</code> – Workitem

Next steps

To use this custom work item, implement custom work queue `MyActivitySetEscalationWorkQueue` as defined in “Implement custom work queue `MyActivitySetEscalationWorkQueue`” on page 648.

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 648.

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
}
}

```

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 657
- “Categorizing a process typecode” on page 660

Custom batch process overview

Custom batch processes are Gosu classes that extend the `BatchProcessBase` base class. The `BatchProcessBase` base class includes code that PolicyCenter 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, PolicyCenter 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 652
- “Categorizing a process typecode” on page 660
- “Implementing `IProcessesPlugin`” on page 661
- *System 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.

Result

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 PolicyCenter 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
<code>DetailStatus</code>	Returns the detailed status for the batch type. Class <code>BatchProcessBase</code> 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 PolicyCenter <code>Administration</code> 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 <code>getDetailStatus</code> , which takes no arguments.
<code>Exclusive</code>	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. 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 Policy 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, <pre>override property get Exclusive() : boolean { return false }</pre> For Java implementations, implement this property getter as the method <code>isExclusive</code> , which takes no arguments.
<code>Finished</code>	Returns a Boolean value to indicate whether the process completed. Completion says nothing about the errors, if any. For Java implementations, implement this property getter as the <code>isFinished</code> method.
<code>OperationsCompleted</code>	Returns a count of how many operations are complete, as an integer value. For Java implementations, implement this property getter as the <code>getOperationsCompleted</code> method.

Property	Description
OperationsExpected	Returns a count of how many operations PolicyCenter expects the batch process to perform, as an integer value. For Java implementations, implement this property as the <code>getOperationsExpected</code> method.
OperationsFailed	Returns an internal count for the number of operations that failed. For Java implementations, implement property getter as the <code>getOperationsFailed</code> method.
Progress	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> . PolicyCenter determines the values of <code>x</code> and <code>y</code> from the <code>OperationsExpected</code> and <code>OperationsCompleted</code> properties. For Java implementations, implement the property getter as the <code>getProgress</code> method.
TerminateRequested	Returns a Boolean value that is the return value from the <code>requestTermination</code> method.
Type	Returns the type of the current batch process. There is no need to override the default <code>BatchProcessBase</code> implementation of this property. For Java implementations, implement the property getter as the <code>getType</code> method, which takes no arguments.

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.

PolicyCenter instantiates batch process classes at startup on servers with the `batch` server role. Later, PolicyCenter calls the `checkInitialConditions` method to determine whether to start a batch process:

- If the method returns `true`, PolicyCenter starts the batch process by calling its `doWork` method.
- If the method returns `false`, the initial conditions are not met and PolicyCenter 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`, PolicyCenter:

- 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 PolicyCenter 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 PolicyCenter 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 PolicyCenter 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, PolicyCenter 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 PolicyCenter 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, PolicyCenter 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. PolicyCenter 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.

PolicyCenter 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 chunkingSize)
```

The `setChunkingById` method returns nothing.

The default PolicyCenter 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 652
- “`incrementOperationsCompleted`” on page 654
- “`incrementOperationsFailed`” on page 654
- *Gosu Reference Guide*

Examples of custom batch processes

This topic contains the following example batch processes:

- “Example batch process for a background task” on page 657
- “Example batch process for unit of work processing” on page 658

See also

- “Developing custom batch process” on page 650

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 638
- 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 typecode 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://PolicyCenter:8180/pc/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" +
    "    <title>${subject}</title>\n" +
    "  </head>\n" +
    "  <body>\n" +
    "    <table>\n" +
    "      <tr><th>Subject</th><th>User</th><th>Group</th><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="PolicyForward.go(actvt.PolicyS);
      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 660
- “Updating the work queue configuration” on page 661
- “Implementing IPProcessesPlugin ” on page 661

The steps to perform depend on whether you implemented a custom work queue or a custom batch process.

See also

- *System 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.

UIRunnable	The process is runnable both from the user interface and from the web service APIs.
APIRunnable	The process is runnable only from web service APIs.
Schedulable	It is possible to schedule the process to execute at specific dates and times.
MaintenanceOnly	It 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 638
- “Define a typecode for a custom batch process” on page 651
- To learn how to add categories to a typecode, see the *Configuration Guide*.

Updating the work queue configuration

PolicyCenter instantiates custom work queues that derive from `WorkQueueBase` or `BulkInsertWorkQueueBase` at start up, for servers in a PolicyCenter cluster with the `workqueue` server role. For PolicyCenter 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 PolicyCenter. 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 PolicyCenter 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 PolicyCenter 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

- System Administration Guide*

Implementing `IProcessesPlugin`

PolicyCenter instantiates custom batch processes that derive from `BatchProcessBase` at startup, for servers in a PolicyCenter cluster with the `batch` server role. Although PolicyCenter instantiates your class on all servers that have the `batch` server role, a single run of your custom batch process executes on only one the servers.

For PolicyCenter 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, PolicyCenter provides a Gosu `ProcessesPlugin` class that implements the `IProcessesPlugin` interface in the following location in Guidewire Studio™ for PolicyCenter:

`configuration→config→gsrc→gw→plugin→processes`

For each typecode in the `BatchProcessType` that represents a custom batch process, PolicyCenter 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 PolicyCenter executes a batch process, whether it is a scheduled process, or requested manually or through web services, PolicyCenter 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({PCConfigParameters.KeepCompletedMessagesForDays.Value})
    }

    construct(arguments : Object[]) {
        super(BatchProcessType.TC_PURGEMESSAGEHISTORY)
        if (arguments.length == 1 and arguments[0] typeis Integer) {
            _ageInDays = Coercions.makeIntFrom(arguments[0])
        } else {
            _ageInDays = PCConfigParameters.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 660
- “Implementing `IProcessesPlugin`” on page 661
- “Using web service methods with batch processes” on page 96
- System Administration Guide*

Monitoring PolicyCenter processes

Although PolicyCenter processes run without human intervention as background tasks, it is important to manage the processes and monitor their progress. PolicyCenter provides a number of features for monitoring default and custom processes.

Monitoring PolicyCenter 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 PolicyCenter 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 PolicyCenter work queues. From this **Work Queue Info** screen, you can also download process history information on individual work queues.

See also

- System Administration Guide*

Monitoring PolicyCenter 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 PolicyCenter 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

- *System 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 96
- *System 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:

- *System 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 PolicyCenter 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`, PolicyCenter 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.

Free-text search integration

PolicyCenter 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 PolicyCenter with the Guidewire Solr Extension.

See also

- *Configuration Guide*
- *Installation Guide*

Free-text search plugins overview

PolicyCenter free-text search depends on two Guidewire plugins that connect PolicyCenter 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 PolicyCenter application.

The free-text search plugin interfaces are:

- **ISolrMessageTransportPlugin** – Called by the Indexing System rules in the Event Fired ruleset whenever policies 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 Policies**→**Basic** search screen to send users' criteria to the Guidewire Solr Extension and receive the search results.

PolicyCenter provides the plugin implementations and the Guidewire Solr Extension software. Do not try connecting the free-text plugins to your own installation of Apache Solr.

IMPORTANT Guidewire does not support replacing the plugin implementations that PolicyCenter 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.

This topic includes the following:

- “Connecting the free-text plugins to the Guidewire Solr Extension” on page 668
- “Enabling and disabling the free-text plugins” on page 668
- “Running the free-text plugins in debug mode” on page 668

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 PolicyCenter. 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 PolicyCenter 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 PolicyCenter. 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 `PCSolrMessageTransport` 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

PolicyCenter provides the free-text load and index message transport to send changed policy data to the Guidewire Solr Extension for loading and incremental indexing. In the base configuration of PolicyCenter, 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.PCSolrMessageTransportPlugin`.
 - 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 `PCSolrMessageTransportPlugin` implementation if you add or remove free-text search fields.

To edit the Plugins Registry item for the `PCSolrMessageTransportPlugin` interface, in the **Project** window in Studio, navigate to **configuration**→**config**→**Plugins**→**registry**, and then open `ISolrMessageTransportPlugin.gwp`.

This topic includes the following:

- “Message destination for free-text search” on page 669
- “Enable the message destination for free-text search” on page 669
- “`ISolrMessageTransportPlugin` plugin parameters” on page 670

See also

- “Messaging and events” on page 309

Message destination for free-text search

The implementation of `PCSolrMessageTransportPlugin` depends on the `PCSolrMessageTransport` 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.PCSolrMessageTransportPlugin`, 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>ContactChanged</code>• <code>ContactAdded</code>• <code>ContactRemoved</code>• <code>JobPurged</code>• <code>PolicyAddressChanged</code>• <code>PolicyPeriodAdded</code>• <code>PolicyPeriodChanged</code>• <code>PolicyPeriodPurged</code>

Field	Value
	<ul style="list-style-type: none"> • PolicyPeriodRemoved • PolicyPurged • PreemptedPeriod
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
<code>commitImmediately</code>	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 PolicyCenter 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>
<code>debug</code>	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

PolicyCenter 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 PolicyCenter, the plugin is disabled. The Plugins Registry specifies the following Gosu implementation:

```
gw.solr.PCSolrSearchPlugin
```

If you add or remove free-text search fields from your configuration, you must modify the implementation of `PCSolrSearchPlugin`.

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
<code>debug</code>	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>

Plugin parameter	Description	Default value
fetchSize	Determines the maximum number of query results that the Guidewire Solr Extension returns for each search request.	100

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 PolicyCenter 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 PolicyCenter application, followed by `/service`, and finally the servlet query path. The following line shows this syntax:

```
http://serverName:8180/pc/service/servletQueryPath
```

You register your servlet with PolicyCenter by adding an entry in the file `PolicyCenter/configuration/config/servlet/servlets.xml`. PolicyCenter 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 PolicyCenter application provides an informational servlet that lists the available servlets. In the base configuration, this servlet is disabled if the PolicyCenter application is in production mode. Use the following URL to display the list:

```
http://serverName:8180/pc/service/info
```

[Viewing servlet analysis information](#)

Guidewire provides a means to view servlet activity in PolicyCenter Profiler. To view this information, navigate to the following location in PolicyCenter Server Tools:

Guidewire Profiler→Configuration

For information on how to use Guidewire Profiler, see the *System Administration Guide*.

Implementing servlets

The PolicyCenter 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 PolicyCenter 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.

PolicyCenter 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.PCLoggerCategory` class.

See also

- “[Implementing servlet authentication](#)” on page 678

@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:8180/pc/service/test  
http://localhost:8180/pc/service/tester  
http://localhost:8180/pc/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:8180/pc/service/test  
http://localhost:8180/pc/service/test0
```

- An entire virtual file hierarchy, such as:

```
http://localhost:8180/pc/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:8180/pc/service/test1/subfolder_one_level_down  
http://localhost:8180/pc/service/test2/subfolder_one_level_down  
http://localhost:8180/pc/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 PolicyCenter 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:8180/pc/service/test0?abc`

RequestURL

The URL that the client used to make the request

Example value: `http://localhost:8180/pc/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: `/pc/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 PolicyCenter. You call this servlet in the context of PolicyCenter.

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:

```
PolicyCenter/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:8180/pc/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 PolicyCenter application.

The following text appears on the page:

```
I am the page /test
```

See also

- “Example of a basic servlet” on page 678
- “Implementing servlet authentication” on page 678

Example of a basic servlet

The following simple Gosu servlet runs in the context of the PolicyCenter 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

@Servlet( \ 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(HttpServletResponse.SC_OK)
        response.getWriter.append("I am the page " + req.PathInfo)
    }
}
```

For PolicyCenter to provide access to the servlet, the `PolicyCenter/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 678

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 `HTTPServlet` 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 PolicyCenter. 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.Writer.append("I am the page " + req.PathInfo)
```

5. Register the servlet class in the file:

```
PolicyCenter/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 680

Test your basic authentication servlet

A servlet can use basic authentication to log in to PolicyCenter. 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 680. To test the servlet, you must use a tool that supports adding HTTP basic authentication headers for the user name and password.

Procedure

1. Run the QuickStart server with the following command:

```
gwb runServer
```

2. Test the servlet at the URL:

```
http://localhost:8180/pc/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 PolicyCenter application.

3. 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: /pc/service/test  
query string: null
```

4. Test additional pages:

```
http://localhost:8180/pc/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: /pc/service/test  
query string: param=value
```

5. 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 PolicyCenter 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:8180/pc/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. PolicyCenter 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 `HTTPServlet` class, refer to this Oracle Javadoc page:

<http://docs.oracle.com/javaee/7/api/javax/servlet/http/HttpServlet.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. PolicyCenter 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

PolicyCenter 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 PolicyCenter 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 PolicyCenter, 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 PolicyCenter. For example, you might use a corporate implementation of single sign-on (SSO) authentication that stores information in HTTP headers other than the HTTP basic headers. You can get the user name and password and call this method. This method forces creation of a new session.</p> <p>In a single sign-on environment, get the current session before calling <code>login</code>. Then, if necessary, do HTTP basic authentication.</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 • Username as a <code>String</code> • Password as a <code>String</code>

See also

- “Example of a servlet using multiple authentication types” on page 683

Example of a servlet using multiple authentication types

The following Gosu servlet runs in the context of the PolicyCenter 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.HttpServlet
uses javax.servlet.http.HttpServletRequest
uses javax.servlet.http.HttpServletResponse
uses javax.servlet.http.HttpServlet

@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.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 PolicyCenter. 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 683.

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 PolicyCenter application.

3. Test the servlet with no authentication by going to the URL:

```
http://localhost:8180/pc/service/test
```

You see a message:

```
HTTP ERROR 401  
Problem accessing /pc/service/test. Reason:  
Unauthorized. No valid user with session context or HTTP Basic.
```

4. Log in to the PolicyCenter application with valid credentials.
5. Test the servlet with the PolicyCenter authenticated user by going to the URL:

```
http://localhost:8180/pc/service/test
```

You see a message:

```
REQUEST SUCCEEDED  
req.RequestURI: '/pc/service/test'  
req.PathInfo: '/test'  
req.RequestURL: 'http://localhost:8180/pc/service/test'  
authenticated user name: 'Super User'
```

6. To test the HTTP basic authentication:
 - a. Sign out of the PolicyCenter application to remove the session context.
 - b. Re-test your servlet, using a tool that supports adding HTTP headers for HTTP basic authentication.

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 45.

Data extraction integration

PolicyCenter provides several mechanisms to generate messages, forms, and letters in custom text-based formats from PolicyCenter data. If an external system needs information from PolicyCenter about a policy, it can send requests to the PolicyCenter 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, PolicyCenter searches for the requested data such as policy 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, PolicyCenter 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 PolicyCenter database. Generate HTML within PolicyCenter or from linked external systems, such as intranet websites that query PolicyCenter and display the results in its own user interface.

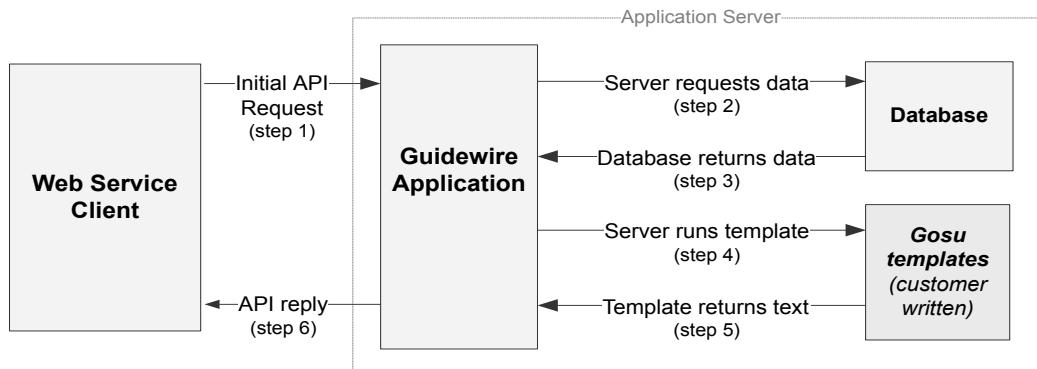
The major techniques to extract data from PolicyCenter by using templates are described below.

- 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 PolicyCenter 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 PolicyCenter 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 policy as a root object for a template, you could design different templates to export policy data, such as the following examples.

- A list of all notes on the policy
- A list of all open activities on the policy
- A summarized view of a policy

To provide programmatic access to PolicyCenter data, PolicyCenter 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 policy, refer to the properties using Gosu expressions as follows.

```
policy.Account  
    Account associated with this policy  
policy.Periods  
    List of policy periods associated with this policy  
policy.PolicyType  
    Policy type
```

The simplest Gosu expressions only extract data object properties, such as `policy.PolicyNumber`. For instance, suppose you want to export the policy number, use the following short template.

```
The number is <%= policy.PolicyNumber %>.
```

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 PolicyCenter 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 all policy revisions associated with a policy. A Gosu template with the root object `policy` might look something like the following code.

```
<% for (var thisPolicyRev in policy.revisions) { %>  
  Policy revision number: <%= thisPolicyRev.PolicyNumber %>  
<% } %>
```

This template might generate something like the following output.

```
Policy revision number: HO-123456-01  
Policy revision number: HO-123456-02  
Policy revision number: HO-123456-03
```

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 `Policy` object, they are available work within Gosu templates just as in business rules within Guidewire Studio. For instance, just use expressions like `myPolicy.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 PolicyCenter log file by using the `gw.api.system.PCLoggerCategory` 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 PolicyCenter typelist mapping tools. PolicyCenter provides access to these tools by using the `$typecode` object.

```
typecode.getAliasByInternalCode("LossType", "ns1", policy.PolicyType)
```

This `$typecode` API works only within Gosu templates, not Gosu in general.

In addition, you can use the `TypecodeMapperUtil` Gosu utility class.

Proxy servers

Guidewire recommends deploying proxy servers to insulate PolicyCenter from the external Internet. This is recommended for any other outgoing requests to computers on the external Internet and to insulate PolicyCenter from incoming requests from the Internet.

Several PolicyCenter integration options require outgoing messages. Placing a proxy server between the external Internet and PolicyCenter insulates PolicyCenter 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 PolicyCenter server directly encrypting HTTPS/SSL connections to an outsider server, PolicyCenter 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 PolicyCenter and not just outgoing requests from PolicyCenter 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 PolicyCenter 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.pgpi.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
Overview of Apache's proxy server module	http://httpd.apache.org/docs/current/mod/mod_proxy.html

Web services and proxy servers

If your PolicyCenter 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 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 PolicyCenter. 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
<code>\$DestinationTrustedCACertFile</code>	File containing the certificate used to sign the destination web site.
<code>\$ReverseProxyTrustedCertFile</code>	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.

Security file	Description
\$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 passphrase. 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 for PolicyCenter

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 PolicyCenter 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 PolicyCenter 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 694

SSL encryption for users

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

See also

- “Upstream (reverse) proxy with encryption for user connections” on page 696
- For information about managing secure communications, see the System 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
SSLCSessionCache shm:$SSL_CACHE(512000)
SSLCSessionCacheTimeout 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
SSLEngine on

#The Reverse proxy checks the destination's certificate
#using the appropriate Trusted CA's certificate
SSLCACertificateFile $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 Hosts 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 PolicyCenter and thus reduce the processing burden of SSL encryption in Java on the PolicyCenter 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 pc directory and its subdirectories for the authorized sites only
    <Directory /pc>
        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 PolicyCenter to respond to a request over SSL from a particular inbound connection, your proxy handles encryption. The connection between PolicyCenter 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 PolicyCenter server is aware of the proxy, edit the web application container server configuration `CATALINA_HOME/conf/server.xml` on your PolicyCenter 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.

<i>port</i>	The port number for the additional connector for access through the proxy
<i>proxyName</i>	The deployment server's name
<i>proxyPort</i>	The port for encrypted access through Apache
<i>scheme</i>	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

PolicyCenter supports the deployment of Java code. There are multiple reasons to write and deploy Java code in PolicyCenter, including accessing entity data. For example, you can implement PolicyCenter 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 PolicyCenter, 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. PolicyCenter does not support adding or modifying Java class files in the Guidewire Studio user interface. Although Studio does not hide user interface tools that add Java classes to the file hierarchies, coding in Java in Studio is unsupported.

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 705
- *Gosu Reference Guide*

Implementing plugin interfaces in Java and optionally OSGi

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

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, PolicyCenter includes an application called IntelliJ IDEA with OSGi Editor.

Guidewire recommends OSGi for all new Java plugin development. PolicyCenter supports OSGi bundles only to implement a PolicyCenter 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 PolicyCenter 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 PolicyCenter 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. PolicyCenter 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 PolicyCenter, 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 PolicyCenter) – A specially-configured instance of IntelliJ IDEA and included with PolicyCenter. 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 PolicyCenter)	--	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 may 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 PolicyCenter)	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	WARNING 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.
`PolicyCenter/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 PolicyCenter data model configuration files. Entity types have built-in properties and you can optionally add extension properties. **Policy** 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 PolicyCenter 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 PolicyCenter 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 PolicyCenter installation directory, run the following command.

```
gwb genJavaApi
```

Result

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

```
PolicyCenter/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.

```
PolicyCenter/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.

```
policy.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_name` 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 PolicyCenter 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.

See also

- *Gosu Reference Guide*

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 PolicyCenter application context. You cannot run that code from your Java IDE because the IDE project does not initialize Gosu. When PolicyCenter 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 `PolicyCenter/modules/configuration/plugins/Gosu/classes` folder. If the interface extends a plugin interface, use the `PolicyCenter/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

PolicyCenter 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.

PolicyCenter 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`, PolicyCenter 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 PolicyCenter class loader delegates Java class loading, PolicyCenter requests the parent class loader to load the class, which is the PolicyCenter application. If the PolicyCenter 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 PolicyCenter 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**.

There are two special values for *PLUGIN_DIR*.

- 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**.

The PolicyCenter/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.

```
PolicyCenter/modules/configuration/plugins/PLUGIN_DIR/classes
```

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

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

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

```
PolicyCenter/modules/configuration/plugins/PLUGIN_DIR/lib
```

Deploy an example Java class with no plugins and no entities

Test the use of Java in your PolicyCenter 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.

```
PolicyCenter/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.

5

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 PolicyCenter Java API libraries with the `getJavaApi` tool. This is a requirement that 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 prompt from the correct application product directory.

Create a new project with an OSGi plugin module

An OSGi plugin module must be in a project. Follow this procedure to create a project.

Before you begin

Before you can create a new project with an OSGi plugin module, you must generate the Java API libraries.

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 already have a project that contains files, you must create a new project. To create a new project, you must do Step 2.

If you have already run IntelliJ IDEA with OSGi Editor and have an empty project that you want to use, do Step 3.

Procedure

1. Launch IntelliJ with OSGi Editor by opening a command prompt in the application root directory and typing the following command.

```
gwb pluginStudio
```

2. 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**.
3. 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 Open the **More Settings** pane, if it is not already open. Set the name of the new module as appropriate. By default, the **Bundle Symbolic Name** field matches the name of the module. The bundle symbolic name defines the main part of the output JAR file name before the version number. You can optionally change the symbolic name to a different value. You can also optionally change the version of the bundle in the **Bundle Version** field..
 - d. To select an existing module to import, choose **Import Module**. Click **Next** repeatedly and confirm the settings for the imported module. Click **Finish**. Click **OK**. You do not do the remaining steps in this procedure.
4. Open the **More Settings** pane, if it is not already open. Set the name of the new module as appropriate. By default, the **Bundle Symbolic Name** field matches the name of the module. The bundle symbolic name defines the main part of the output JAR file name before the version number. You can optionally change the symbolic name to a different value. You can also optionally change the version of the bundle in the **Bundle Version** field.
5. 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.
6. If this is a new project, you must set the project JDK.
 - a. Click **File**→**Project Structure**→**Project Settings**. In the **Project** section, set the **Project SDK** picker to your Java JDK.
 - b. 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.
 - c. Click **OK**.

Next steps

- “Create an OSGi-compliant class that implements a plugin interface” on page 711

Create an OSGi-compliant class that implements a plugin interface

Before you begin

- “Create a new project with an OSGi plugin module” on page 709

Procedure

- Regenerate the Java libraries.
- In IntelliJ IDEA with OSGi Editor, navigate under your OSGi module to the `src` directory.
- Create new subpackages as necessary. Right-click `src` and choose **New→Package**.
The following steps show an example of a simple startable plugin in a `mycompany` package that contains the new classes.
- Right-click the package for your class.
- Choose **New→New OSGi plugin**. IntelliJ IDEA with OSGi Editor opens a dialog.
- In the **Plugin class name** field, enter the name of the Java class that you want to create.
For our example, type `DemoStartablePlugin`.
- 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**.
- IntelliJ IDEA with OSGi Editor displays a dialog **Select Methods to Implement**. By default, all methods are selected. Click **OK**.
- IntelliJ IDEA with OSGi Editor displays your new class with stub versions of all required methods.

10. 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.

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

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.

Example

See “Example startable plugin in Java using OSGi” on page 711.

Next steps

- “Compile and install your OSGi plugin as an OSGi bundle” on page 712

Example startable plugin in Java using OSGi

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

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

```
package mycompany;

import aQute.bnd.annotation.component.Activate;
import aQute.bnd.annotation.component.Component;
import aQute.bnd.annotation.component.ConfigurationPolicy;
import aQute.bnd.annotation.component.Deactivate;
import gw.api.startable.IStartablePlugin;
```

```

import gw.api.startable.StartablePluginCallbackHandler;
import gw.api.startable.StartablePluginState;

import java.util.Map;

@Component(provide = 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;
    }
}

```

The OSGi Ant build script

The main script for OSGi compilation and deployment is an Ant build script. 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 PolicyCenter bundles directory as defined in the OSGi plugin project's `build.properties` file. The script copies your final JAR file to the following PolicyCenter directory.

`PolicyCenter/modules/configuration/deploy/bundles`

Compile and install your OSGi plugin as an OSGi bundle

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

Before you begin

- “Create an OSGi-compliant class that implements a plugin interface” on page 711

Procedure

1. Open a command prompt in the directory that contains your OSGi plugin module.
2. Type the following command.

```
ant install
```

The command generates messages about compiling source files, generating files, copying files, and building a JAR file. If the command succeeds, the following output appears.

```
BUILD SUCCESSFUL
```

3. 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 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 that the script copied the JAR file. If you recently installed a new version of PolicyCenter at a different path, or moved your Guidewire product directory, you must immediately update your OSGi settings in your OSGi project.

4. In 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**. Right-click the item **registry**, and choose **New**→**Plugin**.
- b. In the plugin dialog, enter the plugin name in the **Name** field. For this example, use **DemoStartablePlugin**.

For plugin interfaces that support only one implementation, enter the interface name without the package. For example: **IStartablePlugin**. 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 arbitrary name. 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**.

- c. 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 plugin name **IStartablePlugin**.

- d. Click **OK**.
- e. In the Plugins registry main pane for that **.gwp** file, click the plus sign (+) and select **Add OSGi Plugin**.
- f. 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**.

- g. Set any other fields that your plugin implementation needs, such as plugin parameters.
- h. 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.
- i. Start the server from Studio.

If Studio is already running when you installed or re-installed the bundle, there is an extra required step before running the server. Open a command prompt in the root application directory and run the following command.

```
gwb syncWebApp
```

Next, start the server from Studio, such as by clicking the **Run** or **Debug** menu items or buttons.

Alternatively, you can run the QuickStart server from the command prompt. To do so, open a command prompt in the root PolicyCenter application directory and run the following command.

```
gwb runServer
```

Example

“Example startable plugin in Java using OSGi” on page 711 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 prove that PolicyCenter successfully called your OSGi plugin implementation.

Next steps

- “Configuring third-party libraries in an OSGi plugin” on page 714

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
```

Result

The `ant install` script copies your bundle JAR files to the `PolicyCenter/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 this 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.
- `PolicyCenter/java-api/lib` – The PolicyCenter 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:/PolicyCenter/
```

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.

Inbound files integration

The base configuration of PolicyCenter includes a framework for configuring multiple integrations with external systems. This is achieved by processing file-based data. PolicyCenter 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

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) { }
```

Where:

- `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 *System Administration Guide*.

- `marker` – Marker for logging in `IntentionalLogger`.

InboundFileHandler methods

By default, `BaseInboundFileHandler` implements the following `InboundFileHandler` methods. The methods can be overridden.

Table 10.11-1 InboundFileHandler methods

Method	Parameters	Description
<code>boolean 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.
<code>boolean shouldIgnore(String line, int lineNumber)</code>	<code>line</code> – Raw line as read from the file.	Checks whether to ignore a specific line in a file. For example, a header, a footer, or a comment.

Method	Parameters	Description
<code>boolean isSubRecord(String line, int lineNumber)</code>	<code>lineNumber</code> – Number of the line, starting from 1.	Checks if a line in a file is a sub-record.
<code>void preprocess(IntentionalLogger logger, Marker marker)</code>	<code>line</code> – Raw line as read from the file. <code>logger</code> – IntentionalLogger instance for logging. <code>marker</code> – Marker instance for logging.	Performs pre-processing of the inbound files.
<code>void 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.
<code>void 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.
<code>void 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 InboundFileHandler encounters an exception during processing of an inbound file.
<code>void 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 InboundFileHandler 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 PolicyCenter 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.

Value	Effect
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 *System Administration Guide*.

Configure inbound files integration in the user interface

About this task

Add a configuration in the PolicyCenter 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 PolicyCenter includes a framework that supports creating files for external systems. The files are created from records in the database. PolicyCenter 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 *System 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 10.12-1 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 PolicyCenter 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 *System Administration Guide*.

Configure outbound files integration in the user interface

About this task

Add a configuration in the PolicyCenter 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-java
    javaclass="com.guidewire.outboundfile.plugin.OutboundConfigPlugin">
    <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"?>
<xss:schema targetNamespace="http://guidewire.com/outbound" xmlns:xss="http://www.w3.org/2001/XMLSchema" xmlns="http://guidewire.com/outbound" elementFormDefault="qualified">
  <xss:element name="OutboundConfigurations">
    <xss:complexType>
      <xss:sequence>
        <xss:element type="OutboundLocalConfiguration" name="LocalConfiguration" minOccurs="0" maxOccurs="unbounded" />
        <xss:element type="OutboundS3Configuration" name="S3Configuration" minOccurs="0" maxOccurs="unbounded" />
      </xss:sequence>
    </xss:complexType>
  </xss:element>
  <xss:complexType name="OutboundConfigurationBase">
    <xss:sequence>
      <xss:element type="xs:string" name="Name"/>
      <xss:element type="xs:string" name="Prefix"/>
      <xss:element type="xs:string" name="Extension"/>
      <xss:element type="xs:string" name="FileHandlerClass"/>
      <xss:element type="xs:integer" name="DaysTillPurge"/>
      <xss:element type="xs:string" name="TemporaryDirectory"/>
    </xss:sequence>
    <xss:attribute name="env" type="xs:string"/>
  </xss:complexType>
  <xss:complexType name="OutboundLocalConfiguration">
    <xss:complexContent>
      <xss:extension base="OutboundConfigurationBase">
        <xss:sequence>
          <xss:element type="xs:string" name="PermanentDirectory"/>
        </xss:sequence>
      </xss:extension>
    </xss:complexContent>
  </xss:complexType>
  <xss:complexType name="OutboundS3Configuration">
    <xss:complexContent>
      <xss:extension base="OutboundConfigurationBase">
        <xss:sequence>
          <xss:element type="xs:string" name="ProfileName" minOccurs="0"/>
          <xss:element type="xs:string" name="PermanentBucketName"/>
          <xss:element type="xs:string" name="PermanentPrefix"/>
        </xss:sequence>
      </xss:extension>
    </xss:complexContent>
  </xss:complexType>
</xss: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>
```


Reference specifications

Integration mapping specification

This section contains reference information on integration mappings. This information covers mapping files, mappers, mapping imports, filters, the `JsonMapper` and `TransformResult` classes. The section concludes with a set of integration mapping examples.

Integration mapping overview

What are integration mappings?

Integration mappings are one part of the new integration frameworks defined in Guidewire InsuranceSuite. Moreover, integration mappings are part of an overall feature called 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 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 may 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 738
- “Integration mapping file combination” on page 740

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
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 by schemaName	The name of the schema definition for which this mapper produces output	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	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: <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
mapper	string	No	Either #/mappers/ <name> or <alias>#/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 740

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 761 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 738. 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. PolicyCenter 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 752
- “Integration mapping file specification” on page 738

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 764. 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 747 for an example of how the structure of integration mapping properties and mapper references mimic the structure of JSON schema properties and definition references.

See also

- “Integration mapping file specification” on page 738
- “JSON data types and formats” on page 764
- “Integration mapping examples” on page 747

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 PolicyCenter 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.toPrettyJsonString()
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:

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "definitions": {
    "Address": {
      "type" : "object",
      "properties": {
        "street": {
          "type": "string"
        },
        "city": {
          "type": "string"
        }
      }
    }
  }
}

{
  "schemaName": "contact-1.0",
  "mappers": [
    {
      "Address": {
        "schemaDefinition" : "Address",
        "transform": {
          "method": "getAddress"
        }
      }
    }
  ]
}
```

```

"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"
        },
        "subtype": {
            "type": "string",
            "x-gw-type": "typekey.Contact"
        }
    }
}
}

"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).FirstName",
            "predicate" : "Contact typeis Person"
        },
        "primaryAddress": {
            "path" : "Contact.PrimaryAddress",
            "mapper": "#/mappers/Address"
        },
        "subtype": {
            "path" : "Contact.Subtype"
        }
    }
}
}

```

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 752
- “JSON schema file combination” on page 761
- <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	http://json-schema.org/draft-04/schema#	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

Property	Type	Required	Format	Behavior	Combination Style
definitions	map<string, <i>Definition Object</i> >	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, <i>Property Object</i> >	No	The string keys in the map serve as the names of the properties.	Defines the properties that this schema definition explicitly names. If a JSON object includes one of these specified properties, the object must match the type and constraints set forth in the associated property definition.	Merged by key
additionalProperties	<i>Property Object</i>	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. PolicyCenter reports a validation error only if the required property is undefined on the JSON object.	Merge names
x-gw-extensions	map<string, any>	No	This property is an arbitrary map of property keys to values. The	Values in extensions are available to IRestValidatorFactoryPlugin objects for use in building custom validations.	Merge extensions

Property	Type	Required	Format	Behavior	Combination Style
				keys can be any string. The values can be any object, including JSON objects or arrays.	

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

Property	Type	Required	Format	Behavior	Combination Style
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. PolicyCenter parses the value of the maximum property as a fixed-point value with no loss of precision. PolicyCenter 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. On the one hand, suppose that the value of the type property is integer and the value of the format property is int32. In this case, the value of the maximum property must be a valid integer. PolicyCenter thus converts the value of the maximum property to an Integer at runtime. On the other hand, suppose that the respective values of the type and format properties are string and gw-bigdecimal. In this case, PolicyCenter converts the value of the maximum property to a BigDecimal for comparison at runtime. Note that PolicyCenter permits the maximum property to have a value only if the runtime type is numeric.	First non-null
exclusiveMaximum	boolean	No		Determines if the maximum value is treated as inclusive or exclusive. The default value of the exclusiveMaximum property is false. This means that the value of the maximum property is inclusive. PolicyCenter permits the exclusiveMaximum property to have a value only if you specify the maximum property.	First non-null
minimum	number	No		Specifies the minimum value the property can have. Otherwise, the minimum property behaves in the same way as the maximum property. Note that PolicyCenter permits the minimum property to have a value only if the runtime type is numeric.	First non-null

Property	Type	Required	Format	Behavior	Combination Style
exclusiveMinimum	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. PolicyCenter permits the <code>exclusiveMinimum</code> property to have a value only if you specify the <code>maximum</code> property.	First non-null
maxLength	integer	No		Determines the maximum length of the property value. The value of the maximum length is inclusive. For PolicyCenter 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
minLength	integer	No		Determines the minimum length of the property value. The value of the minimum length is inclusive. For PolicyCenter 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
pattern	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, PolicyCenter evaluates the regular expression using the Java regular expression engine. The regular expression generally matches the Java syntax. This syntax has minor differences from the Javascript syntax and therefore represents a slight deviation from the JSON Schema specification. For PolicyCenter to permit the <code>pattern</code> property to have a value, the property must have a runtime type of <code>String</code> .	First non-null
maxItems	integer	No		Specifies the maximum number of elements allowed in an array. This value is inclusive. PolicyCenter permits <code>maxItems</code> for properties only of type <code>array</code> .	First non-null
minItems	integer	No		Specifies the minimum number of elements allowed in an array. This value is inclusive. PolicyCenter permits <code>minItems</code> for properties only of type <code>array</code> .	First non-null
uniqueItems	boolean	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> . PolicyCenter permits <code>uniqueItems</code> for properties only of type <code>array</code> .	First non-null

Property	Type	Required	Format	Behavior	Combination Style
multipleOf	number	No		<p>Specifies that the property value must be a multiple of the specified value. PolicyCenter parses the value of the <code>multipleOf</code> property as a fixed-point value with no loss of precision. PolicyCenter 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. PolicyCenter 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, PolicyCenter converts the value of the <code>multipleOf</code> property to a <code>BigDecimal</code> for comparison at runtime.</p> <p>Note that PolicyCenter permits the <code>multipleOf</code> property to have a value only if the runtime type is numeric.</p>	First non-null
readOnly	boolean	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
enum	any[]	No		Each element must be a JSON value that can be parsed based on the <code>type</code> , <code>format</code> , and <code>x-gw-type</code> of this property.	First non-null
x-gw-xml	Property XML Object	No		Allows configuration of how PolicyCenter maps the property definition into an XSD.	First non-null
x-gw-export-enumeration	boolean	No		If the value of this property is <code>true</code> , PolicyCenter writes typekey values out as an <code>enum</code> property while creating a schema for external clients. This property is only valid if <code>enum</code> is not set and <code>x-gw-type</code> is a typekey type. The default value is <code>false</code> .	First non-null
x-gw-nullable	boolean	No		Specifies that this property can have an explicit <code>null</code> value. This property defaults to <code>false</code> . A value of <code>false</code> means that the property must either be undefined on the object or must have an explicit non-null value.	First non-null

Property	Type	Required	Format	Behavior	Combination Style
x-gw-extensions	map<string, any>	No	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 use in building custom validations.	Merge extensions

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 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
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	First non-null

Property	Type	Required	Format	Behavior	Combination Style
				maximum property in the property object table.	
exclusiveMaximum	boolean	No		See the behavior for the <code>exclusiveMaximum</code> 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 <code>minimum</code> property is identical to the behavior defined for the <code>minimum</code> property in the property object table.	First non-null
exclusiveMinimum	boolean	No		See the behavior for the <code>exclusiveMinimum</code> property in the property object table.	First non-null
maxLength	integer	No		Determines the maximum length of property items. Otherwise, the behavior for the <code>maxLength</code> property is identical to the behavior defined for the <code>maxLength</code> property in the property object table.	First non-null
minLength	integer	No		Determines the minimum length of property items. Otherwise, the behavior for the <code>minLength</code> property is identical to the behavior defined for the <code>minLength</code> 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 <code>pattern</code> property is identical to the behavior defined for the <code>pattern</code> 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 <code>multipleOf</code> property is identical to the behavior defined for the <code>pattern</code> 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 <code>type</code> , <code>format</code> , and <code>x-gw-type</code> of this item.	Specifies a list of values at least one of which item values must match. PolicyCenter turns the <code>enum</code> values in the schema into Java objects at runtime. PolicyCenter then compare the <code>enum</code> values against the item value using the <code>equals</code> and <code>hashCode</code> methods.	First non-null
x-gw-xml	Items XML Object	No		Allows configuration of how PolicyCenter maps the item definition into an XSD.	First non-null
x-gw-export-enumeration	boolean	No		If the value of this property is <code>true</code> , PolicyCenter writes typekey values out as an <code>enum</code> property if creating a schema for external clients. This property is only valid if <code>enum</code> is not set and <code>x-gw-type</code> is a typekey type. The default value is <code>false</code> .	First non-null
x-gw-nullable	boolean	No		Specifies that this property can have items with an explicit <code>null</code> value. This property defaults to <code>false</code> .	First non-null

Property	Type	Required	Format	Behavior	Combination Style
x-gw-extensions	map<string, any>	No	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 use in building custom validations.	Merge extensions

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 761
- “JSON data types and formats” on page 764
- <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 752. 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 752

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 PolicyCenter adopts such formats where they are available. PolicyCenter 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.

PolicyCenter 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 PolicyCenter maps a property for serialization and deserialization. PolicyCenter 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 PolicyCenter can serialize, the actual serialization format, and the Java type to which PolicyCenter deserializes the input data.

Types

The following table lists all the combinations of `type`, `format`, and `x-gw-type` that PolicyCenter 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
<code>string</code>	<code>gw-money</code>	<code>gw.pl.currency.MonetaryAmount</code>	Guidewire
<code>string</code>	<code>gw-money</code>	<code>gw.api.financials.CurrencyAmount</code>	Guidewire

Type	Format	x-gw-type	Standard
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 PolicyCenter can serialize to a given format. The **Java Output** column lists the output type to which PolicyCenter 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, 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, PolicyCenter 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
<code>Boolean</code>	<code>boolean</code>	<code><none></code>	<code><none></code>
<code>Integer</code>	<code>integer</code>	<code>int32</code>	<code><none></code>
<code>Long</code>	<code>integer</code>	<code>int64</code>	<code><none></code>
<code>Float</code>	<code>number</code>	<code>float</code>	<code><none></code>
<code>Double</code>	<code>number</code>	<code>double</code>	<code><none></code>
<code>String</code>	<code>string</code>	<code><none></code>	<code><none></code>
<code>Date</code>	<code>string</code>	<code>date-time</code>	<code><none></code>
<code>LocalDate</code>	<code>string</code>	<code>date</code>	<code><none></code>
<code>LocalTime</code>	<code>string</code>	<code>time</code>	<code><none></code>
<code>BigDecimal</code>	<code>string</code>	<code>gw-bigdecimal</code>	<code><none></code>
<code>BigInteger</code>	<code>string</code>	<code>gw-biginteger</code>	<code><none></code>
<code>CurrencyAmount</code>	<code>string</code>	<code>money</code>	<code>gw.api.financials.CurrencyAmount</code>
<code>MonetaryAmount</code>	<code>string</code>	<code>money</code>	<code>gw.pl.currency.MonetaryAmount</code>
<code>Blob</code>	<code>string</code>	<code>byte</code>	<code><none></code>
<code>byte[]</code>	<code>string</code>	<code>byte</code>	<code><none></code>
any TypeKey type	<code>string</code>	<code><none></code>	<code>typekey.<name></code>

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 `JsonObject`s.
- 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 764.
- It deserializes array properties into lists of `JsonObject`s 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 752. 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 764. 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 JSON 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 764, 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, PolicyCenter 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>	<code>false</code>	If <code>true</code> , null property values are preserved. If <code>false</code> , null properties are undefined in the output.
<code>includeNullItems</code>	<code>false</code>	If <code>true</code> , the REST framework preserves the null item values in arrays. If <code>false</code> , 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>	<code>true</code>	If <code>true</code> , empty arrays for properties are preserved. If <code>false</code> , empty arrays are treated as if the property value was <code>null</code> , subject to the value of the <code>includeNullProperties</code> option.
<code>includeSchemaProperty</code>	<code>false</code>	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>	<code>false</code>	Currently ignored for <code>JsonObject</code> serialization.
<code>includeTimestamp</code>	<code>false</code>	Currently ignored for <code>JsonObject</code> serialization.
<code>includeUser</code>	<code>false</code>	Currently ignored for <code>JsonObject</code> serialization.
<code>includeServer</code>	<code>false</code>	Currently ignored for <code>JsonObject</code> serialization.
<code>validateOutput</code>	<code>false</code>	If <code>true</code> , 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 766 and “JSON serialization” on page 768 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.pc.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

PolicyCenter generates the schema wrapper classes in Java under the `jsonschema` namespace. PolicyCenter 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 769. 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 framework 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",  
          "x-gw-type": "typekey.OfficialIDValue"  
        }  
      }  
    }  
  }  
}
```

```
        "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:xss="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"/>
<xss:element
    name="IntegerExt"
    type="xs:int"
    minOccurs="0"/>
<xss:element
    name="LastName"
    type="xs:string"
    minOccurs="0"/>
<xss:element
    name="OfficialIDs"
    minOccurs="0">
    <xss:complexType>
        <xss:sequence>
            <xss:element
                name="OfficialID"
                type="OfficialID"
                minOccurs="0"
                maxOccurs="unbounded"/>
        </xss:sequence>
    </xss:complexType>
</xss:element>
<xss:element
    name="PrimaryAddress"
    type="Address"
    minOccurs="0"/>
<xss:element
    name="TagTypes"
    minOccurs="0">
    <xss:complexType>
        <xss:sequence>
            <xss:element
                name="TagTypes"
                type="xs:string"
                minOccurs="0"
                maxOccurs="unbounded"
                gw:type="typekey.ContactTagType"/>
        </xss:sequence>
    </xss:complexType>
</xss:element>
</xss:all>
</xss:complexType>
<xss:complexType
    name="OfficialID">
    <xss:all>
        <xss:element
            name="Jurisdiction"
            type="xs:string"
            minOccurs="0"
            gw:type="typekey.Jurisdiction"/>
        <xss:element
            name="Type"
            type="xs:string"
            minOccurs="0"
            gw:type="typekey.OfficialIDType"/>
        <xss:element
            name="Value"
            type="xs:string"
            minOccurs="0"/>
    </xss:all>
</xss:complexType>
<xss:complexType
    name="TagMap">
    <xss:sequence>
        <xss:any
            minOccurs="0"
            maxOccurs="unbounded"
            processContents="skip"/>
    </xss:sequence>
</xss:complexType>
```

```

</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"/>
<xs:element
  name="OfficialID"
  type="OfficialID"/>
<xs:element
  name="TagMap"
  type="TagMap"/>
</xs: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 PolicyCenter 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, PolicyCenter 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.pc.activities.activities-1.0
```

This code adds the `activities-1.0` JSON schema from the `pc` 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.

