

Guidewire PolicyCenter®

PolicyCenter Integration Guide

RELEASE 8.0.4

Copyright © 2001-2015 Guidewire Software, Inc. All rights reserved.

Guidewire, Guidewire Software, Guidewire ClaimCenter, Guidewire PolicyCenter, Guidewire BillingCenter, Guidewire Reinsurance Management, Guidewire ContactManager, Guidewire Vendor Data Management, Guidewire Client Data Management, Guidewire Rating Management, Guidewire InsuranceSuite, Guidewire ContactCenter, Guidewire Studio, Guidewire Product Designer, Guidewire Live, Guidewire DataHub, Guidewire InfoCenter, Guidewire Standard Reporting, Guidewire ExampleCenter, Guidewire Account Manager Portal, Guidewire Claim Portal, Guidewire Policyholder Portal, Gosu, Deliver Insurance Your Way, and the Guidewire logo are trademarks, service marks, or registered trademarks of Guidewire Software, Inc. in the United States and/or other countries.

All other trademarks are the property of their respective owners.

This material is confidential and proprietary to Guidewire and subject to the confidentiality terms in the applicable license agreement and/or separate nondisclosure agreement.

Guidewire products are protected by one or more United States patents.

Product Name: Guidewire PolicyCenter

Product Release: 8.0.4

Document Name: PolicyCenter Integration Guide

Document Revision: 25-May-2015

Contents

About PolicyCenter Documentation	19
Conventions in This Document	20
Support	20

Part I Planning Integration Projects

1 Integration Overview.....	23
Overview of Integration Methods.....	23
PolicyCenter Integration Elements	27
Important Information about PolicyCenter Web Services	28
Preparing for Integration Development	28
Integration Documentation Overview	29
API Reference Javadoc Documentation	29
Gosu Generated Documentation	30
Using Javadoc-formatted Documentation.....	30
Data Dictionary Documentation	31
Regenerating Integration Libraries and WSDL	31
What are Required Files for Integration Programmers?.....	32
Public IDs and Integration Code.....	33
Creating Your Own Public IDs.....	34

Part II Web Services

2 Web Services Introduction.....	37
What are Web Services?.....	37
Publish or Consume Web Services from Gosu.....	37
Finding the Best Web Service Documentation for Your Needs.....	38
What Happens During a Web Service Call?.....	38
Reference of All Built-in Web Services	39
3 Publishing Web Services	41
Web Service Publishing Overview	42
Designing Your Web Services	42
Committing Entity Data to the Database Using a Bundle	44
Serializable Gosu Classes Must Be Final and Exportable	44
Web Service Publishing Quick Reference	44
Web Service Publishing Annotation Reference	45
Publishing and Configuring a Web Service	46
Declaring the Namespace for a Web Service	46
Specifying Minimum Run Level for a Web Service.....	47
Specifying Required Permissions for a Web Service	47
Overriding a Web Service Method Name or Visibility	48
Web Service Invocation Context	48
Web Service Class Lifecycle and Request Local Scoped Variables.....	50
Testing Web Services with Local WSDL.....	50
Writing Unit Tests for Your Web Service	51

Generating WSDL	52
Getting WSDL from a Running Server.....	52
Generating WSDL On Disk	54
Special Annotation to Generate WSDL On Disk	54
Command Line tool to Generate WSDL On Disk	54
Generating SOAP APIs for Use with Unit Tests From Gosu.....	55
Adding Advanced Security Layers to a Web Service.....	55
Data Stream Transformations	55
Web Services Authentication Plugin	59
The Default Web Services Authentication Plugin Implementation	60
Writing an Implementation of the Web Services Authentication Plugin	60
Checking for Duplicate External Transaction IDs	61
Request or Response XML Structural Transformations.....	61
Reference Additional Schemas in Your Published WSDL	62
Validate Requests Using Additional Schemas as Parse Options	62
Invocation Handlers for Implementing Preexisting WSDL	63
Adding an Invocation Handler for Preexisting WSDL.....	63
Example of an Invocation Handler for Preexisting WSDL	64
Invocation Handler Responsibilities	65
Locale Support	66
Setting Response Serialization Options, Including Encodings.....	67
Exposing Typelists and Enumerations as String Values	67
Transforming a Generated Schema	68
Login Authentication Confirmation	68
Stateful Session Affinity Using Cookies	69
Calling a PolicyCenter Web Service from Java	69
Calling Web Services using Java 1.6 and wsimport	70
Adding HTTP Basic Authentication in Java.....	71
Adding SOAP Header Authentication in Java	72
4 Calling Web Services from Gosu.....	75
Consuming Web Service Overview	75
Loading WSDL Locally Using Studio Web Service Collections.....	76
Loading WSDL Directly into the File System	76
How Does Gosu Process WSDL?.....	78
Web Service API Objects Are Not Thread Safe	79
Learning Gosu XML APIs	79
What Gosu Creates from Your WSDL	80
A Real Example: Weather.....	81
Request XML Complexity Affects Appearance of Method Arguments.....	81
Adding Configuration Options	82
HTTP Authentication	84
Guidewire Authentication.....	84
Suite Configuration File Sets URLs to Guidewire Applications	84
Setting Guidewire Transaction IDs	85
Setting a Timeout	86
Custom SOAP Headers	86
Server Override URL	87
Setting XML Serialization Options.....	87
Setting Locale in a Guidewire Application.....	88
Implementing Advanced Web Service Security with WSS4J	88
One-Way Methods	89
Asynchronous Methods	90
MTOM Attachments with Gosu as Web Service Client.....	91

5 General Web Services	93
Mapping Typecodes to External System Codes	93
Using Web Services to Translate Typecodes	94
Using Gosu or Java to Translate Typecodes	94
Importing Administrative Data	95
CSV Import and Conversion	95
Advanced Import or Export	95
Maintenance Tools Web Service	96
Running Batch Processes Using Web Services	96
Manipulating Work Queues Using Web Services	96
Stopping Startable Plugins Using Web Services	97
System Tools Web Services	97
Getting and Setting the Run Level	97
Getting Server and Schema Versions	98
Workflow Web Services	98
Profiling Web Services	99
6 Account and Policy Web Services	101
Account Web Services	102
Job Web Services	104
Common Parameters in the Job APIs	104
Helper Methods of the Job APIs	105
Policy Cancellation and Reinstatement Web Services	105
Beginning a Cancellation	105
Rescinding a Cancellation	106
Finding a Cancellation	106
Reinstating a Policy	107
Submission Web Services	107
Producer Web Services	108
Product Model Web Services	109
Synchronize Product Model in Database with File System XML	109
Synchronize System Tables	110
Query the Database for Product Model Information	110
Policy Web Services	111
Adding a Referral Reason	111
Closing a Referral Reason	111
Add Activity from Pattern	111
Policy Period Web Services	112
Adding Notes to Policies and Policy Periods	113
Adding Documents to Policy Periods	113
Policy Earned Premium Web Services	113
Import Policy Web Services	113
Policy Change Web Services	114
Starting an Automatic Policy Change Job	114
Starting a Manual Policy Change Job	114
Specifying What Policy Data to Change	114
Policy Renewal Web Services	115
Starting Renewals on Existing Policies	115
Importing a Policy for Renewal	115
Policy Renewal Methods for Billing Systems	118
Archiving Web Services	119
Check If a Policy Term is Archived	119
Restore a Policy Term	119
Suspending and Resuming Archiving	119

Quote Purging Web Services	119
Do Not Purge Flag on a Policy Period	120

Part III Plugins

7 Plugin Overview	123
Overview of PolicyCenter Plugins	124
Implementing Plugin Interfaces	124
Built-in Plugin Implementation Classes	126
Registering a Plugin Implementation Class	126
Deploying Java Files (Including Java Code Called From Gosu Plugins).....	128
Additional Information By Plugin Type.....	129
Error Handling in Plugins.....	129
Temporarily Disabling a Plugin	129
Example Gosu Plugin	129
Special Notes For Java Plugins	130
Which Libraries to Compile Java Code Against.....	130
Where To Put Java Class and Library Files Needed By Java Plugins	131
Useful Java Plugin APIs	131
Getting Plugin Parameters from the Plugins Registry Editor.....	131
Writing Plugin Templates For Plugins That Take Template Data.....	132
Plugin Registry APIs	134
Plugin Thread Safety	135
Avoid Singletons Due to Thread-Safety Issues	138
Design Plugin Implementations to Support Server Clusters.....	139
Reading System Properties in Plugins	139
Do Not Call Local Web Services From Plugins.....	140
Creating Unique Numbers in a Sequence.....	140
Restarting and Testing Tips for Plugin Developers	141
Summary of All PolicyCenter Plugins	141
8 Account and Policy Plugins.....	149
Policy Number Generator Plugin	150
Policy Numbers Conform to Field Validator	150
Account Plugin Details	151
Job Process Creation Plugin	152
Job Number Generator Plugin	152
Policy Term Plugin	153
Policy Period Plugin	154
Setting Written Date for a Transaction	155
Customizing Behavior After Applying Changes from a Branch	155
Specifying Types to Omit If Copying a Contractual Period.....	155
Customizing Behavior After Setting Period Window	156
Customizing Behavior After Creating Draft Branch In New Period	156
Customizing Behavior Before Promoting a Branch	156
Customizing Behavior After Handling a Preemption	156
Loss History Plugin	157
Location Plugin.....	157
Comparing Locations	157
Customizing Location Cloning	158
Policy Plugin.....	158
Effective Time Plugin.....	161
Policy Payment Plugin	162

Underwriting Company Plugin	163
Policy Evaluation Plugin	164
Renewal Plugin	165
Notification Plugin	166
Reference Date Plugin	168
Audit Schedule Selector Plugin	168
Proration Plugin	169
Leap Days and Proration	169
Provide Your Own Prorater Subclass	170
Motor Vehicle Record (MVR) Plugin	171
Overview of Motor Vehicle Record Integration	171
How PolicyCenter Uses Motor Vehicle Records	171
Implementations of the Motor Vehicle Record Plugin	172
Writing a Motor Vehicle Record Plugin	172
Policy Hold Job Evaluation Plugin	173
Quote Purging Plugin	173
Quote Purging Plugin Methods and Getters	174
Purge Context Object	177
ETL Product Model Loader Plugin	177
ETL Product Model Loader Plugin Implementation	178
Non-operational Plugin Implementation	178
Conversion on Renewal Plugin	178
9 Authentication Integration	181
Overview of User Authentication Interfaces	181
User Authentication Source Creator Plugin	183
User Authentication Service Plugin	185
Example Authentication Service Authentication	188
Deploying User Authentication Plugins	188
Database Authentication Plugins	189
ContactManager Authentication	190
10 Document Management	191
Document Management Overview	191
Document Storage Overview	192
Document Production Overview	192
Document Retrieval Mode Overview	192
Choices for Storing Document Content and Metadata	193
Deciding Where to Store Document Content and Metadata	193
Internal Versus External Metadata Permanently Affects Some Objects	194
Document Storage Plugin Architecture	195
Implementing a Document Content Source for External DMS	196
Adding Documents and Metadata	196
Retrieving Documents	197
Checking for Document Existence	197
Removing Documents	198
Updating Documents and Metadata	198
Storing Document Metadata In an External DMS	199
The Built-in Document Storage Plugins	200
Built-in Document Storage Directory and File Name Patterns	200
Remember to Store Public IDs in the External System	201
Asynchronous Document Storage	201
Configuring Asynchronous Document Storage	202

APIs to Attach Documents to Business Objects.....	202
Gosu APIs to Attach Documents to Business Objects	202
Web Service APIs to Attach Documents to Business Objects	203
Retrieval and Rendering of PDF or Other Input Stream Data	203
11 Document Production.....	205
Document Production Overview	205
User Interface Flow for Document Production.....	206
Document Production Plugins	210
Configuring Document Production and MIME Types	211
Document Template Descriptors	213
Template Descriptor Fields for Metadata About Each Template	214
Template Descriptor Fields and Defaults for Document Metadata	216
Template Descriptor Fields and Context Objects	216
Template Descriptor Fields Related to Form Fields and Values to Merge.....	218
XML Format of Built-in IDocumentTemplateSerializer	219
Template Source Reference Implementation	223
Built-in Document Production Plugins.....	224
Generating Documents from Gosu	224
Important Notes About Cached Document UIDs.....	226
Template Web Service APIs.....	226
12 Geographic Data Integration	229
Geocoding Plugin Integration.....	229
How PolicyCenter Uses Geocode Data.....	230
What the Geocoding Plugin Does.....	230
Synchronous and Asynchronous Calls to the Geocoding Plugin	230
Using a Proxy Server with the Geocoding Plugin	230
The Built-in Bing Maps Geocoding Plugin	231
Batch Geocoding Only Some Addresses	231
Steps to Deploy a Geocoding Plugin	231
Step 1: Implement the Plugin Interface in Gosu.....	231
Step 2: Register the Plugin Implementation in Studio	231
Step 3: Enable the User Interface for Desired Geocoding Features	231
Writing a Geocoding Plugin	232
Using the Abstract Geocode Java Class	232
High-Level Steps to Writing a Geocoding Plugin Implementation	232
Geocoding an Address	233
Getting Driving Directions	235
Getting a Map for an Address.....	237
Getting an Address from Coordinates (Reverse Geocoding)	237
Geocoding Status Codes.....	237
13 Encryption Integration	239
Encryption Integration Overview	239
Setting Encrypted Properties.....	240
Querying Encrypted Properties	241
Writing Your Encryption Plugin.....	241
Installing Your Encryption Plugin	243
Adding or Removing Encrypted Properties	244
Set Encryption Plugin for Encrypted Properties in Archived Objects	244
Changing Your Encryption Algorithm Later	244
Changing Your Encryption Algorithm	245
14 Management Integration.....	247
Management Integration Overview	247

The Abstract Management Plugin Interface	248
Integrating With the Included JMX Management Plugin.....	249
15 Other Plugin Interfaces.....	251
Territory Code Plugin	251
Vehicle Identification Number Plugin	253
Automatic Address Completion and Fill-in Plugin	253
Phone Number Normalizer Plugin	254
Testing Clock Plugin (Only For Non-Production Servers)	254
Using the Testing Clock Plugin	254
Testing Clock Plugin in PolicyCenter Clusters.....	255
Work Item Priority Plugin	255
Official IDs Mapped to Tax IDs Plugin	255
Preupdate Handler Plugin.....	256
Defining Base URLs for Fully-Qualified Domain Names	256
Exception and Escalation Plugins.....	257
16 Startable Plugins.....	259
Startable Plugins Overview	259
Startable Plugins as Background Processes	260
Registering Startable Plugins	260
Starting, Stopping, and Managing Startable Plugins	260
Startable Plugins in a Clustered Configuration.....	260
Writing a Startable Plugin.....	260
User Contexts for Startable Plugins	262
Simple Startable Plugin Example	262
Startable Plugins and Run Levels	263
Configuring Startable Plugins to Run on All Servers.....	263
Java and Startable Plugins	266
Persistence and Startable Plugins	266
17 Multi-threaded Inbound Integration.....	267
Multi-threaded Inbound Integration Overview	267
Inbound Integration Configuration XML File	267
Inbound Integration Core Plugin Interfaces	268
Inbound Integration Handlers for File and JMS Integrations	268
Inbound Integration Configuration XML File	269
Thread Pool Configuration	269
Configuring a List of Inbound Integrations	270
Inbound File Integration	271
Example File Integration.....	275
Inbound JMS Integration	276
Custom Inbound Integrations	278
Writing a Work Agent Implementation	279
Installing a New Custom Inbound Integration	283
Understanding the Polling Interval and Throttle Interval.....	285

Part IV Messaging

18 Messaging and Events	289
Messaging Overview	290
Event	290
Message	290
Message History	290
Messaging Destinations	291
Root Object	291
Primary Entity and Primary Object	291
Acknowledgement (ACK and NAK)	292
Safe Ordering	292
Transport Neutrality	293
Messaging Flow Overview	293
Messaging Flow Details	295
Restrictions on Entity Data in Messaging Rules and Messaging Plugins	299
Messaging Interacts With PolicyCenter Workflow	300
Database Transactions When Creating Messages	301
Messaging in PolicyCenter Clusters	301
Messaging Plugins Must Not Call SOAP APIs on the Same Server	301
Message Destination Overview	302
Handling Acknowledgements	304
Messaging Database Transactions During Sending	308
Built-In Destinations	308
Filtering Events	309
Validity and Rule-Based Event Filtering	309
List of Messaging Events in PolicyCenter	309
Triggering Custom Event Names	316
No Events from Import Tool	317
Generating New Messages in Event Fired Rules	317
Saving Intermediate Values Across Rule Set Executions	320
Creating a Payload Using Gosu Templates	320
Setting a Message Root Object or Primary Object	321
Creating XML Payloads Using Guidewire XML (GX) Models	322
Using Java Code to Generate Messages	323
Saving Attributes of the Message	323
Maximum Message Size	323
Message Ordering and Multi-Threaded Sending	323
Ordering Non-safe-ordered Messages	324
If Multiple Events Fire, Which Message Sends First?	324
Message Ordering and Performance Tuning Details	325
Late Binding Data in Your Payload	328
Reporting Acknowledgements and Errors	329
Message Sending Error Behaviors	329
Submitting Acks, Errors, and Duplicates from Messaging Plugins	330
Tracking a Specific Entity With a Message	333
Implementing Messaging Plugins	333
Getting Message Transport Parameters from the Plugin Registry	333
Implementing a Message Request Plugin	334
Implementing a Message Transport Plugin	334
Implementing a Message Reply Plugin	336
Error Handling in Messaging Plugins	338
Handling Messaging Destination Suspend, Resume, Shutdown	339

Resynchronizing Messages for a Primary Object.....	339
How Resync Affects Pre-Update and Validation.....	342
Resync in ContactManager.....	342
Message Payload Mapping Utility for Java Plugins.....	342
Message Status Code Reference.....	343
Monitoring Messages and Handling Errors	344
Error Handling Concepts.....	344
The Administration User Interface	346
Web Services for Handling Messaging Errors	346
Messaging Tools Web Service	347
Batch Mode Integration	348
Included Messaging Transports	349
The Built-in Email Transport	349
Enabling the Built-in Console Transport	349

Part V Policy-related Integrations

19 Rating Integration.....	353
The Rating Framework	353
Guidewire Rating Management and PCRatingPlugin	357
Overview of Cost Data Objects.....	358
Where to Override the Default Rating Engine?	360
Common Questions About the Default Rating Engine.....	362
Optional Asynchronous Rating.....	366
Implementing Rating for a New Line of Business	367
What Do Cost Data Objects Contain?.....	368
Cost Core Properties	368
Adding Line-specific Cost Properties and Methods	374
Fixed ID Keys Link a Cost Data Object to Another Object.....	375
Cost Data Object Methods and Constructors to Override.....	376
Cost Data APIs That You Can Call	382
Checklist for Relationship Changes in Cost Data Objects	383
Writing Your Own Line-specific Rating Engine Subclass.....	384
Rating Line Example for Personal Auto.....	393
A Close Look at PersonalAutoCovCostData	394
Rating Variations	397
Workers' Compensation Rating	398
Inland Marine Rating	400
General Liability Rating	400
20 Reinsurance Integration.....	403
Reinsurance Integration Overview	403
Reinsurance Data Model.....	405
Risk Entity.....	405
Risk Version List Entity	406
Reinsurance Plugin	407
Architecture of the Reinsurance Management Plugin	408
Configuration Plugins for Guidewire Reinsurance Management.....	408
Creating a Plugin Implementation for Your Own Reinsurance System.....	409
Reinsurance Plugin Implementation	409
Reinsurance Plugin Interface Methods	410

Reinsurance Configuration Plugin	413
Reinsurance Configuration Plugin Implementations	413
Reinsurance Configuration Plugin Methods and Properties.....	413
Reinsurance Ceding Plugin.....	416
Reinsurance Ceding Plugin Implementations	416
Reinsurance Ceding Plugin Methods	417
Reinsurance Coverage Web Service.....	418
Methods of the PolicyCenter Reinsurance Coverage Web Service	419
21 Forms Integration	423
Forms Integration Overview	423
Forms Inferences Classes	424
Creating Inference Data and XML	424
Determining Whether to Add or Reprint a Form	426
Form Data Helper Functions	427
Handling Multiple Instances of One Form	428
Reference Dates on a Form.....	429
Forms Messaging	429
21 Policy Difference Customization	431
Policy Difference Overview	431
Customizing How to Display Difference Items to Users	434
Difference Item Subclasses.....	435
Difference Helper Classes.....	435
Difference Utility Classes	436
Compare Individual Objects with Matchers	437
Important Files for Customizing Differences	440
Filtering Difference Items (After Database Generation)	441
Difference Tree XML Configuration	441
Editing the Difference Tree XML Files	444
Customizing Differences for New Lines of Business.....	449
Customizing Personal Auto Line of Business	450
APIs for Calculating Differences	451
Differences Between a Branch at its Based-on Branch	452
Differences Between any Two Branches	452

Part VI

Billing, Claim, and Contact Integrations

22 Billing Integration	455
Billing Integration Overview	456
Mechanisms for PolicyCenter and BillingCenter Integration.....	456
Integrating Policy Center with Another Billing System.....	457
How Billing Data Flows Between Applications.....	457
Notifying a Billing System of Policy Changes and Premiums	458
Tracking Policies Term by Term	458
Some PolicyCenter Properties Are Mainly for Billing Systems.....	458
Asynchronous Communication.....	459
Exit Points Between Applications.....	461
Configuring Which System Receives Contact Updates	461
Billing Producers and Producer Codes	462
Billing Accounts	464
Billing Instructions in BillingCenter.....	467
Billing Instruction Subtypes	467

Billing Flow for New-Period Jobs	470
Flow of Submission, Renewal, and Rewrite.....	470
Policy Period Mapping	472
Billing Methods and Payment Plans.....	473
New Periods and Term Confirmed Flag.....	473
Billing Flow for Existing-Period Jobs	473
Billing Implications of Midterm Changes	474
Billing Implications of Renewals or Rewrites	476
Multiple Choices of Renewal Flow	477
Account Creation for Conversion on Renewal.....	482
Copying Billing Data to New Periods on Renewal or Rewrite.....	482
Billing Implications for Cancellations and Reinstatements	483
Cancellations That Start in PolicyCenter	483
Cancellations That Start in BillingCenter.....	483
Billing Implications of Audits	484
Billing Implications for Premium Reporting	486
Billing Implications of Delinquency for Failure to Report.....	487
Billing Implications of Deposits	487
Deposits in Submission, Renewal, or Rewrite Jobs	488
Deposits in Policy Change and Reinstatement Jobs	489
Deposits in Premium Reports	489
Deposits in Cancellation	489
Deposits in Final Audit.....	490
Implementing the Billing System Plugin	490
Account Management.....	491
Policy Period Management.....	492
Get Period Information from Billing System	493
Billing System Notifications of PolicyCenter Policy Actions	494
Producer Management	496
Agency Bill Plan Availability Retrieval.....	497
Commission Plan Management	498
Payment Plans and Installment Previews	498
Updating Contacts.....	499
Implementing the Billing Summary Plugin	500
Payment Integration	502
Use Integration-Specific Containers for Integration	504
23 Multicurrency Integration between BillingCenter and PolicyCenter	505
Set up Currencies for Multicurrency Integration	505
Configure Account Numbers for Multicurrency Accounts in BillingCenter	506
Affiliated Account Numbers and Billing Communication	506
Assign Account Numbers to Affiliated Multicurrency Accounts.....	506
24 Claim and Policy Integration	507
Claim Search from PolicyCenter	507
Policy System Notifications	509
Policy Search Web Service (For Claim System Integration)	509
Typecode Maximum Length and Trimmed Typecodes	511
Policy Search SOAP API	511
PolicyCenter Exit Points to ClaimCenter and BillingCenter	511
PolicyCenter Product Model Import into ClaimCenter	512
Configuring the ClaimCenter Typelist Generator.....	514
Running the ClaimCenter Typelist Generator.....	517
Using Generated Typelists in ClaimCenter.....	518
Typelist Localization.....	522

Policy Location Search API	523
Dependencies of the Policy Location Search API	523
Finding Policy Locations within Geographic Bounding Boxes	523
25 Contact Integration.....	525
Integrating with a Contact Management System	525
Asynchronous Messaging with the Contact Management System	526
Contact Retrieval.....	527
Contact Searching	528
Support for Finding Duplicates.....	529
Finding Duplicates	529
Adding Contacts to the External System	530
Updating Contacts in the External System	531
Overwrite the Local Contact with Latest Values	531
Configuring How PolicyCenter Handles Contacts	531
Synchronizing Contacts with Accounts	532
Account Contact Plugin	533
Account Contact Role Plugin	533
Contact Web Service APIs	534
Delete a Contact	534
Determine if a Contact Can Be Deleted	535
Add Contact	535
Update Contacts	536
Merging Contact Addresses	536
Merge Contacts	537
Handling Rejection and Approval of Pending Changes	537
Activating Contacts.....	538
Get Associated Work Orders for a Contact.....	538
Get Associated Policies for a Contact.....	539
Get Associated Accounts for a Contact	539
Address APIs	540
Updating an Address.....	540

Part VII

Importing Policy Data

26 Zone Import.....	543
Introduction to Zone Import	543
Staging Tables.....	544
Zone Data	544
Integrity Checks	545
Load Error Tables	545
Load History Tables	545
Load Commands and Loadable Entities	545
Overview of a Typical Database Staging Table Import	546
Importing Zone Data.....	546
High-Level Steps in a Typical Database Staging Table Import	547
Detailed Steps in a Typical Database Staging Table Import.....	547
Database Import Performance and Statistics	550
Table Import Tools	550
Other Import-Related Tools	551
Data Integrity Checks	552
Table Import Tips and Troubleshooting	553

Part VIII Other Integration Topics

27 Archiving Integration	557
Overview of Archiving Integration.....	557
Archiving Integration and Archiving Eligibility Flow	558
Archive Source Plugin	560
Archiving Eligibility Plugin	561
Archiving Web Services in PolicyCenter	561
Set Encryption Plugin for Encrypted Properties in Archived Objects	561
Archiving Storage Integration Detailed Flow.....	562
Archive Source Plugin Storage Methods	563
Archive Retrieval Integration Detailed Flow	565
Archive Source Plugin Retrieve Methods.....	566
Archive Source Plugin Utility Methods	566
Check for Archiving Before Accessing Policy Data	568
Upgrading the Data Model of Retrieved Data	568
28 Custom Batch Processing	569
Overview of Custom Batch Processing	569
Styles of Batch Processing	570
Choosing a Style for Custom Batch Processing	570
Nightly and Daytime Batch Processing	570
Batch Processing Typecodes.....	571
Developing Custom Work Queues	572
Custom Work Queue Overview	572
Defining the Typecode for Your Custom Work Queue	574
Defining the Work Item Type for Your Custom Work Queue.....	574
Creating Your Custom Work Queue Class.....	575
Developing the Writer for Your Custom Work Queue.....	576
Developing the Workers for Your Custom Work Queue	577
Example Work Queues.....	579
Simple Example of a Work Queue	579
Example Work Queue for Updating Entities	580
Example Work Queue with a Custom Work Item Type.....	581
Developing Custom Batch Processes	582
Custom Batch Process Overview	583
Creating a Custom Batch Process	583
Batch Process Implementation Using the Batch Process Base Class	584
Example Batch Processes	586
Example Batch Process for a Background Task	587
Example Batch Process for Unit of Work Processing	587
Enabling Custom Batch Processing to Run	589
Categorizing Your Batch Processing Typecode	590
Updating the Work Queue Configuration.....	590
Implementing the Processes Plugin	591
Monitoring Batch Processing	591
The Work Queue Info Page.....	591
The Batch Process Info Page.....	592
Monitoring for Batch Processing Completion	592
Maintenance Tools	592
Process History	592
Periodic Purging of Batch Processing Entities	593

29 Free-text Search Integration	595
Free-text Search Plugins Overview	595
Connecting the Free-text Plugins with the Guidewire Solr Extension	596
Enabling and Disabling the Free-text Plugins.....	596
Running the Free-text Plugins in Debug Mode	596
Free-text Load and Index Plugin and Message Transport	596
Message Destination for Free-text Search	597
Plugin Parameters	598
Free-text Search Plugin.....	598
30 Servlets	601
Implementing Servlets	601
Creating a Basic Servlet	601
Implementing Servlet Authentication.....	603
Alternative APIs for Authentication	606
31 Data Extraction Integration	609
Why Gosu Templates are Useful for Data Extraction	609
Data Extraction Using Web Services	610
Error Handling in Templates.....	611
Getting Parameters from URLs.....	611
Built in Templates.....	611
Using Loops in Templates	611
Structured Export Formats	612
Handling Data Model Extensions in Gosu	612
Gosu Template APIs Common for Integration.....	612
32 Logging	615
Logging Overview For Integration Developers	615
Logging Elements	615
Logging Types: Category-based and Class-based	616
Logging Properties File	616
Logging APIs for Java Integration Developers	617
Category-based Logging	617
Logger Classes	617
Class-based Logging (Not Generally Recommended)	618
Dynamically Changing Logging Levels	618
33 Proxy Servers	621
Proxy Server Overview.....	621
Resources for Understanding and Implementing SSL	622
Web Services and Proxy Servers	622
Configuring a Proxy Server with Apache HTTP Server	622
Apache Basic Installation Checklist	622
Certificates, Private Keys, and Passphrase Scripts.....	623
Proxy Server Integration Types for PolicyCenter.....	623
Proxy Building Blocks	624
Downstream Proxy With No Encryption	625
Downstream Proxy With Encryption	625
Upstream (Reverse) Proxy with Encryption for Service Connections	626
Upstream (Reverse) Proxy with Encryption for User Connections	626

34 Java and OSGi Support	629
Overview of Java and OSGi Support	629
Learning More About Entity Java APIs	630
Accessing Gosu Types from Java	630
Implementing Plugin Interfaces in Java and Optionally OSGi	630
Inspections to Flag Unsupported Internal Java APIs	631
Accessing Entity and Typecode Data in Java	633
Regenerating Java API Libraries	633
Entity Packages and Customer Extensions from Java	634
Typecode Classes from Java	638
Comparing Entity Instances and Typecodes	641
Entity Bundles and Transactions from Java	641
Creating New Entity Instances from Java	642
Getting and Setting Entity Properties from Java	643
Calling Entity Object Methods from Java	643
Querying for Entity Data in Java	644
Accessing Gosu Classes from Java Using Reflection	644
Gosu Enhancement Properties and Methods in Java	645
Class Loading and Delegation for non-OSGi Java	645
Java Class Loading Rules	645
Java Class Delegate Loading	646
Deploying Non-OSGi Java Classes and JARs	646
OSGi Plugin Deployment with IntelliJ IDEA with OSGi Editor	647
Generate Java API Libraries	647
Launch IntelliJ IDEA with OSGi Editor	647
Create an OSGi-compliant Class that Implements a Plugin Interface	648
Compiling and Installing Your OSGi Plugin as an OSGi Bundle	649
Using Third-Party Libraries in Your OSGi Plugin	651
Advanced OSGi Dependency and Settings Configuration	654
Updating Your OSGi Plugin Project After Product Location Changes	655

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 how to upgrade PolicyCenter from a previous major version. The intended readers are system administrators and implementation engineers who must merge base application changes into existing PolicyCenter application extensions and integrations.
<i>New and Changed Guide</i>	Describes new features and changes from prior PolicyCenter versions. Intended readers are business users and system administrators who want an overview of new features and changes to features. Consult the "Release Notes Archive" part of this document for changes in prior maintenance releases.
<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. The intended readers are all IT staff and 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 PolicyCenter Studio. The intended readers are business analysts who define business processes, as well as programmers who write business rules in Gosu.
<i>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>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>Glossary</i>	Defines industry terminology and technical terms in Guidewire documentation. The intended readers are everyone who works with Guidewire applications.

Document	Purpose
<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.

Conventions in This Document

Text style	Meaning	Examples
<i>italic</i>	Emphasis, special terminology, or a book title.	A <i>destination</i> sends messages to an external system.
bold	Strong emphasis within standard text or table text.	You must define this property.
narrow bold	The name of a user interface element, such as a button name, a menu item name, or a tab name.	Next, click Submit .
<code>monospaced</code>	Literal text that you can type into code, computer output, class names, URLs, code examples, parameter names, string literals, and other objects that might appear in programming code. In code blocks, bold formatting highlights relevant sections to notice or to configure.	Get the field from the <code>Address</code> object.
<code>monospaced italic</code>	Parameter names or other variable placeholder text within URLs or other code snippets.	Use <code>getName(first, last)</code> . <code>http://SERVERNAME/a.html</code> .

Support

For assistance, visit the Guidewire Resource Portal – <http://guidewire.custhelp.com>

part I

Planning Integration Projects

Integration Overview

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.

This topic includes:

- “Overview of Integration Methods” on page 23
- “PolicyCenter Integration Elements” on page 27
- “Important Information about PolicyCenter Web Services” on page 28
- “Preparing for Integration Development” on page 28
- “Integration Documentation Overview” on page 29
- “Regenerating Integration Libraries and WSDL” on page 31
- “What are Required Files for Integration Programmers?” on page 32
- “Public IDs and Integration Code” on page 33

Overview of Integration Methods

PolicyCenter addresses the following integration architecture requirements:

- **A service-oriented architecture** – Encapsulate your integration code so upgrading the core application requires few other changes. Also, a service-oriented architecture allows APIs to use different languages or platform.
- **Customize 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.
- **Send messages to external systems in a transaction-safe way** – Trigger actions after important events happen inside PolicyCenter, and notify external systems if and 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.
- **Link 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.
- **Import or export data to/from external systems** – There are many methods of importing and exporting data to PolicyCenter, and you can choose which methods make the most sense for your integration project.
- **Use clearly-defined industry standard protocols for integration points** – PolicyCenter includes built-in 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 includes multiple methods of integrating external code with the PolicyCenter product. The primary integration methods:

- **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 would be to programmatically add submissions, policy revisions, and policies into PolicyCenter. 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 easily call web services hosted on other computers to trigger actions or retrieve data.
- **Plugins** – PolicyCenter plugins are mini-programs that PolicyCenter invokes to perform an action or calculate a result. Guidewire recommends writing plugins in Gosu. You can also write plugins in Java. Note that Gosu code can call third-party Java classes and Java libraries.

General categories of plugins include:

- **Messaging plugins** – Send messages to remote systems, and receive acknowledgements 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 batch server, in a separate thread and database transaction, sends messages and tracks (waits for) message acknowledgments. For details, see “Messaging and Events” on page 289.
- **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. For details, see “Authentication Integration” on page 181.
- **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. For details, see “Document Management” on page 191.
- **Inbound integration plugins** – Multi-threaded inbound integrations for high-performance data import. PolicyCenter includes built-in implementations for reading files and receiving JMS messages, but you can also write your implementations that leverage the multi-threaded framework. See “Multi-threaded Inbound Integration” on page 267.
- **Other plugins** – For example, generate policy numbers (`IPolicyNumGenAdapter`), or get a claim related to a policy from a claims system (`IClaimSearchAdapter`). For more information, see “Summary of All PolicyCenter Plugins” on page 141.

See also

- “Plugin Overview” on page 123
- **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. For details, see “Data Extraction Integration” on page 609.

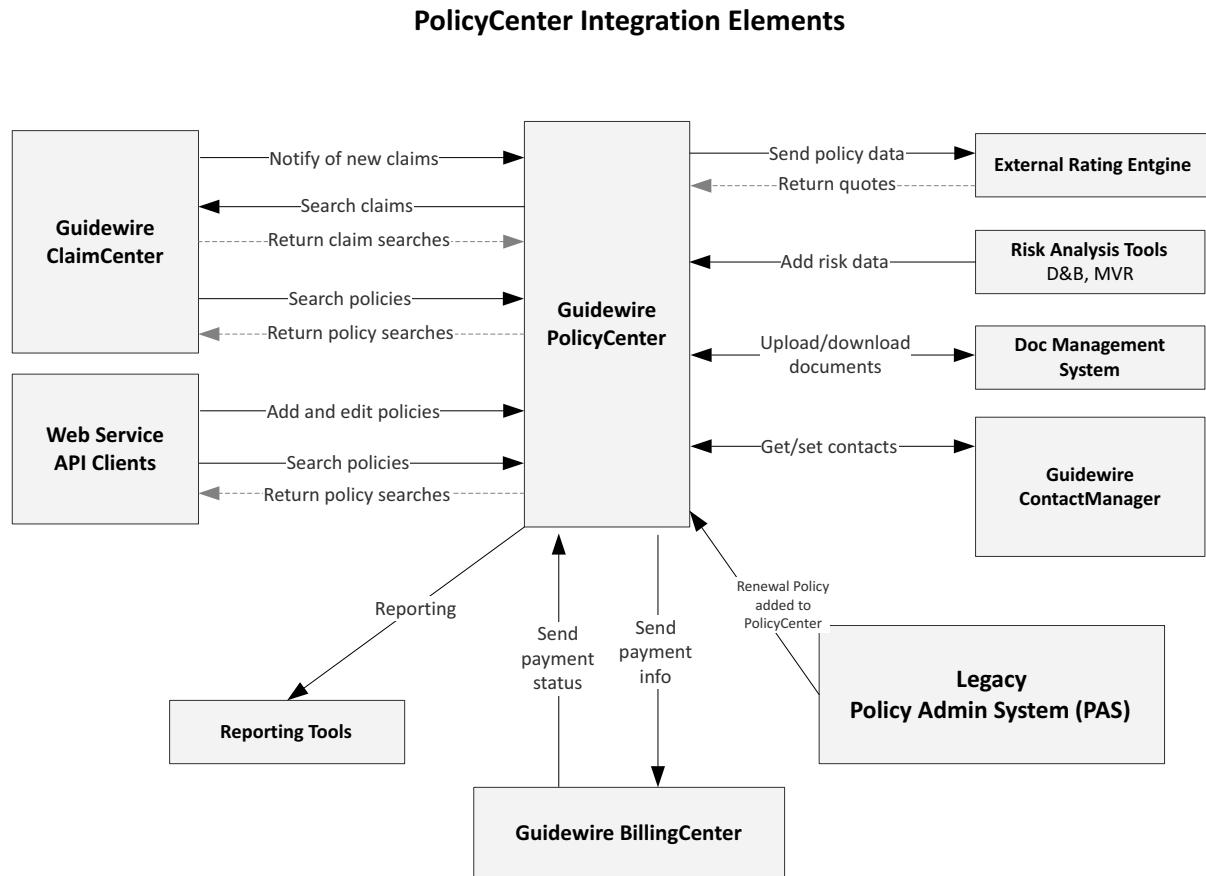
The following table compares the main integration methods.

Integration type	Description	What you write
Web services	A full API to manipulate PolicyCenter data and trigger actions externally using any programming language or platform using PolicyCenter Web services APIs, accessed 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 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 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 using XSDs.
Plugins	Small programs that PolicyCenter calls to perform tasks or calculate a result. Plugins run in the same Java virtual machine (JVM) as PolicyCenter.	<ul style="list-style-type: none"> Gosu classes that implement a Guidewire-defined interface. Java classes that implement a Guidewire-defined interface. Optionally use OSGi, a Java component system. See “Overview of Java and OSGi Support” on page 629
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 acknowledgements.	<ul style="list-style-type: none"> Gosu code in the Event Fired rule set. Messaging plugins, most importantly the MessageTransport plugin. There is also the MessageRequest plugin (for pre-processing your payload) and the MessageReply plugin (for asynchronous replies). Configure one or more messaging destinations in Studio to use 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 are Data Transfer Objects (DTOs). Define DTOs as Gosu classes or using XSDs.

Integration type	Description	What you write
Guidewire XML (GX) models	The GX model editor in Studio lets you create custom schema definitions (XSDs) for your data to assist in integrations. You customize which properties in an entity (or other type) to export in XML. Then, you can use this model to export XML or import XML in your integrations. For example, your messaging plugins could send XML to external systems, or your web services could take this XML format as its payload from an external system. For more information, see “The Guidewire XML (GX) Modeler” on page 306 in the <i>Gosu Reference Guide</i> .	<ul style="list-style-type: none">Using the GX model editor in Studio, you customize which properties in an entity (or other type) to export in XML.Various integration code that uses the XSD it creates.
Templates	Several data extraction mechanisms that perform database queries and format the data as necessary. For example, send notifications as form letters and use plain text with embedded Gosu code to simplify deployment and ongoing updates. Or design template that exports HTML for easy development of web-based reports of PolicyCenter data.	<ul style="list-style-type: none">Text files that contain small amounts of Gosu code

PolicyCenter Integration Elements

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



See also

- For information about integration with BillingCenter, see “Billing Integration” on page 455.
- For information about integration with ContactManager, see “Managing Integrated Contacts” on page 37 in the *Contact Management Guide*.
- For information about integration with ClaimCenter, see “Claim and Policy Integration” on page 507.

Important Information about PolicyCenter Web Services

PolicyCenter supports WS-I web services. WS-I web services use the SOAP protocol and are compatible with the WS-I standard. For an overview of web services, including a list of all built-in web services, see “Web Services Introduction” on page 37.

Topic	See
Overview of web services	“Web Services Introduction” on page 37
Reference of all built-in web services	“Reference of All Built-in Web Services” on page 39
Publishing a web service on the PolicyCenter server	“Publishing Web Services” on page 41
Consuming a web service	“Calling Web Services from Gosu” on page 75

Preparing for Integration Development

During integration development, edit configuration files within 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 Configuration 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`

This directory contains subdirectories such as:

Directory under configuration module	Description
<code>config/web</code>	Your web application PCF files.
<code>config/logging</code>	Your logging configuration files. See “Logging” on page 615.
<code>config/templates</code>	This directory contains two types of templates relevant for integration: <ul style="list-style-type: none"> Plugin templates – Use Gosu plugin templates for the small number of plugin interfaces that require them. These plugin templates extract important properties from entities, and they generate text that describes the results. Whenever PolicyCenter needs to call the plugin, PolicyCenter passes the results to the plugin as text-based parameters. Messaging templates – Use optional Gosu messaging templates for your messaging code. Use messaging templates to define notification letters or other similar messages contain large amounts of text but a small amount of Gosu code.

There are built-in versions some of these templates. To modify the built-in versions in Studio, navigate to Resources → Configuration → Other Resources → Templates.

Directory under configuration module	Description
config/docmgmt	Document management files. See "Document Management" on page 191.
plugins	<p>Your Java plugin files. To add Java files for plugin support, see "Special Notes For Java Plugins" on page 130 and "Overview of Java and OSGi Support" on page 629.</p> <p>Register your plugin implementations in the Plugin registry in Studio. When you register the plugin in the Plugin registry, you can define a <i>plugin directory</i>, which is the name of a subdirectory of the plugins directory. If you do not define a subdirectory, PolicyCenter defaults the plugin directory to the shared subdirectory. Be aware that if you are using the Java API introduced in version 8.0, your classes and libraries must be in a subdirectory of the plugin directory called basic. See "Deploying Non-OSGi Java Classes and JARs" on page 646.</p> <p>For more information about registering a plugin, see "Plugin Overview" on page 123 and "Using the Plugins Registry Editor" on page 131 in the <i>Configuration Guide</i>.</p> <p>For a messaging plugin, you must register this information 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. See "Messaging and Events" on page 289 and "Messaging Editor" on page 153 in the <i>Configuration Guide</i>.

There are some important directories other than the subdirectories for files in the configuration module.

Other directory	Description
PolicyCenter/bin	<p>Command line tools such as gwpc.bat. Use this for the following integration tasks:</p> <ul style="list-style-type: none"> • Regenerating the Java API libraries and web service (SOAP) WSDL. See "Regenerating Integration Libraries and WSDL" on page 31. • Regenerating the <i>Data Dictionary</i>. See "Data Dictionary Documentation" on page 31
PolicyCenter/soap-api	<p>Files related to web services. PolicyCenter scripts generate this directory. To regenerate these, see "Regenerating Integration Libraries and WSDL" on page 31.</p>
PolicyCenter/soap-api/wsi/wsdl	<p>For web services, this directory contains WSDL files generated locally. To regenerate these, see "Regenerating Integration Libraries and WSDL" on page 31.</p>
PolicyCenter/admin	<p>Command line tools that control a running PolicyCenter server. Almost all of these are small Gosu scripts that call public web service APIs.</p>

Integration Documentation Overview

Use the *Integration Guide* for important integration information. Also, consult other reference documentation during development:

- *API Reference Javadoc Documentation*
- *Data Dictionary Documentation*

API Reference Javadoc Documentation

The easiest way to learn what interfaces are available is by reading *PolicyCenter API Reference Javadoc* documentation, which is web browser-based documentation.

There are multiple types of reference documentation.

Java API Reference Javadoc

The Java API Javadoc includes:

- The specification of the plugin definitions for Java plugin interfaces. These also are the specifications for plugins implemented in Gosu.
- The details of full entities, including both the entity data (get and set methods) plus additional methods called domain methods. For a high-level overview of entity differences and different entity access types, see the diagram in “What are Required Files for Integration Programmers?” on page 32. For writing Java plugins, also see “Calling Java from Gosu” on page 123 in the *Gosu Reference Guide*.
- General Java utility classes.
- After regenerating the generated files, find it in `PolicyCenter/java-api/doc/api/index.html`

To regenerate these files, see “Regenerating Integration Libraries and WSDL” on page 31.

Web Service API Reference Documentation

Web Service Description Language (WSDL)

On a running PolicyCenter server, you can get up-to-date WSDL from published services. See “Getting WSDL from a Running Server” on page 52

After regenerating SOAP API reference files, there are additional local WSDL files on the server. To regenerate these files, see “Regenerating Integration Libraries and WSDL” on page 31. PolicyCenter generates the WSDL at the path

```
PolicyCenter/soap-api/wsi/wsdl
```

For more information about web services, see “Web Services Introduction” on page 37.

Web Service Javadoc

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

For more information about web services, see “Web Services Introduction” on page 37.

Gosu Generated Documentation

Integration programmers might want to use the Gosu documentation that you can generate from the command line using the gwpc tool:

```
gwpc regen-gosudoc
```

You will then find the documentation at `PolicyCenter/build/gosudoc/index.html`.

This documentation is particularly valuable for integration programmers implementing plugins in Gosu. The information in the Javadoc-formatted files are more relevant for Gosu programmers than Java programmers due to package differences and visibility from Java.

For more information, see “Gosu Generated Documentation (Gosudoc)” on page 38 in the *Gosu Reference Guide*.

Using Javadoc-formatted Documentation

Several types of generated documentation are in web-based Javadoc format. For Javadoc-formatted documentation, to look within a particular package namespace, click in the top-left pane of the documentation on the package name. The bottom left pane displays interfaces and classes listed for that package. If you click on a class, the right pane shows the methods for that class.

Data Dictionary Documentation

Another set of documentation called the *PolicyCenter Data Dictionary* provides documentation on classes that correspond to PolicyCenter data. You must regenerate the *Data Dictionary* to use it.

To view the documentation, refer to:

```
PolicyCenter/build/dictionary/data
```

To regenerate the dictionary, open a command window and change directory to `PolicyCenter/bin` and run the command:

```
gwpc regen-dictionary
```

The *PolicyCenter Data Dictionary* typically has more information about data-related objects than the API Reference documentation has for that class/entity. The *Data Dictionary* documents only classes corresponding to data defined in the data model. It does not document all API functions or utility classes.

The *Data Dictionary* lists some objects or properties that are part of the data model and the relevant database tables as internal properties. You must not modify internal properties within the database or use any other mechanism. Getting or setting internal data properties might provide misleading data or corrupt the integrity of application data.

For example, the `PolicyPeriod` class's `status` property.

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. Next, you can recompile integration code against the generated libraries, if necessary.

- The location for the Java generated libraries is:
`PolicyCenter/java-api/lib`
- The location of the generated Java library documentation:
`PolicyCenter/java-api/doc`
- The location for the WS-I web service WSDL is:
`PolicyCenter/soap-api/wsi/wsdl`

Call the main `gwpc.bat` file to regenerate these

- For Java development, generate libraries and documentation with:
`gwpc.bat regen-java-api`
- For web service (SOAP) development, generate WSDL using the command:
`gwpc.bat regen-soap-api`

For example, to generate Java API libraries and documentation:

1. In Windows, open a command window.
2. Change your working directory with the following command:
`cd PolicyCenter/bin`
3. Type the command:
`gwpc regen-java-api`

As part of its normal behavior while regenerating the documentation for the Java or the SOAP files, the script displays some warnings and errors.

See also

- “What are Required Files for Integration Programmers?” on page 32

What are Required Files for Integration Programmers?

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

There are several types of integration files:

- **Web services** – Some files represent files you can use with SOAP API client code.
- **Plugin interfaces** – Some files represent plugin interfaces for your code to respond to requests from PolicyCenter to calculate a value or perform a task.
- **Java API libraries** – Some files represent *entities*, which are PolicyCenter objects defined in the data model with built-in properties and can also have data model extension properties. For example, `Policy` and `Address` are examples of Guidewire entities. To access entity data and methods from Java, you need to use Java API libraries. See “Java and OSGi Support” on page 629.

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 may be additional methods available on the objects. These additional *domain methods* often contain valuable functionality for you. If an integration point can access both entity data and domain methods, it is said to have access to the *full entities*.

The following table summarizes the different entity integration implications for each integration type. For this table, *full entity instances* means access to all properties and domain methods.

Entity access	Description	Entities	Necessary libraries
Gosu plugin implementation	Plugin interface defined in Gosu.	Full entity instances	None
Java plugin implementation	Java code that accesses an entity associated with a plugin interface parameter or return value.	Full entity instances	Java API libraries

Entity access	Description	Entities	Necessary 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 are the newer standard for web services.	<p>There is no support for entity types as arguments or return values. Instead, create your own data transfer objects (DTO) as either:</p> <ul style="list-style-type: none"> • Gosu class instances that contain only the properties required for each integration point. • 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. See "The Guidewire XML (GX) Modeler" on page 306 in the <i>Gosu Reference Guide</i>. 	There is no support for entity types as arguments or return values.

See also

- “Web Services Introduction” on page 37
- “Plugin Overview” on page 123

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.

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 20 character public ID limit.

See also

- “`PublicIDPrefix`” on page 55 in the *Configuration Guide*

part II

Web Services

Web Services Introduction

You can write web service APIs in Gosu and access them from remote systems using the standard web services protocol *SOAP*, the standard Simple Object Access Protocol. Web services provide a language-neutral, platform-neutral mechanism for invoking actions or requesting data from other applications across a network. 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 publishes its own built-in web service APIs that you can use.

This topic includes:

- “What are Web Services?” on page 37
- “What Happens During a Web Service Call?” on page 38
- “Reference of All Built-in Web Services” on page 39

What are Web Services?

Web services define request-and-response APIs that let you call an API on a remote computer, or even the current computer, using an abstracted well-defined interface. A data format called the Web Service Description Language (WSDL) describes available web services that other systems can call 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.

Publish or Consume Web Services from Gosu

Gosu natively supports web services in two ways:

- **Publish your Gosu code as new web service APIs** – Write Gosu code that external systems call as a web service using the SOAP protocol. Simply add a single line of code before the definition of a Gosu class.
- **Call web service APIs that external applications publish from your Gosu code** – Write Gosu code that imports and calls web service APIs published by external systems. Gosu parses the 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 with a natural Gosu syntax.

Finding the Best Web Service Documentation for Your Needs

PolicyCenter supports WS-I web services. WS-I web services use the SOAP protocol and are compatible with the WS-I standard. For an overview of web services, including a list of all built-in web services, see “Web Services Introduction” on page 37.

Topic	See
Overview of web services	“Web Services Introduction” on page 37
Reference of all built-in web services	“Reference of All Built-in Web Services” on page 39
Publishing a web service on the PolicyCenter server	“Publishing Web Services” on page 41
Consuming a web service	“Calling Web Services from Gosu” on page 75

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*.

For example:

- Suppose you write a web service in Gosu and 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. PolicyCenter serializes it into a text-based reply to the remote system.
- Suppose you use Gosu to call a web service hosted by an external system. Before calling the API, Gosu automatically 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 elegantly 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, trigger actions, or to change data within the PolicyCenter database.

Publishing a web service can be as simple as adding one special 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.

See also

- For more details about publishing or consuming web services, see “Finding the Best Web Service Documentation for Your Needs” on page 38.

Reference of All Built-in Web Services

The following table lists all built-in 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 105.
JobAPI	Adds an activity to a job. See “Job Web Services” on page 104.
PolicyChangeAPI	Starts a policy change job. See “Policy Change Web Services” on page 114.
PolicyRenewalAPI	Starts a renewal job. See “Policy Renewal Web Services” on page 115.
ReinstatementAPI	Starts a reinstatement job for a policy that has been cancelled. See “Reinstating a Policy” on page 107.
SubmissionAPI	Starts a submission job. See “Submission Web Services” on page 107.
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 102.
AddressAPI	Notifies PolicyCenter of updates to addresses from an external contact system. See “Address APIs” on page 540.
ArchiveAPI	manipulate archiving from external systems. See “Archiving Web Services” on page 119.
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 509.
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 509.
ContactAPI	Notifies PolicyCenter of changes to contacts, including merging of contacts and merging of contact addresses. See “Contact Web Service APIs” on page 534.
ImportPolicyAPI	Adds an activity to a policy. See “Import Policy Web Services” on page 113.
PolicyAPI	Adds an activity to a policy. See “Policy Web Services” on page 111.
PolicyEarnedPremiumAPI	Calculates earned premiums for a policy number as of a specific date. See “Policy Earned Premium Web Services” on page 113.
ProductModelAPI	Modifies the product model on a running development (non-production) server. See “Product Model Web Services” on page 109.
PolicyLocationSearchAPI	Retrieve a list of policy location summaries within a rectangular geographic bounding box. See “Policy Location Search API” on page 523.
PolicyPeriodAPI	Performs various actions on a policy period, such as adding a note or a document. See “Policy Period Web Services” on page 112.
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 509.
ProducerAPI	Performs various actions on producers, agencies, and branches. See “Producer Web Services” on page 108.
ProductModelAPI	Modifies the Product Model on a development (non-production) server. See “Product Model Web Services” on page 109.
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 418.

Web Service Name	Description
General web services	
TypeListToolsAPI	Retrieves aliases for PolicyCenter typecodes in external systems. See “Mapping Typecodes to External System Codes” on page 93.
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.
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.
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 Services” on page 97.
WorkflowAPI	Performs various actions on a workflow, such as suspending and resuming workflows and invoking workflow triggers. See “Workflow Web Services” on page 98.
ProfilerAPI	Sends information to the built-in system profiler. See “Profiling Web Services” on page 99.
MessagingToolsAPI	Manages the messaging system remotely for message acknowledgements error recovery. The methods of this web service are available only when the server run level is multiuser. See “Web Services for Handling Messaging Errors” on page 346.
DataChangeAPI	Tool for rare cases of mission-critical data updates on running production systems. See “Data Change API Overview” on page 49 in the <i>System Administration Guide</i> .
TemplateToolsAPI	Lists and validates Gosu templates available on the server. See “Template Web Service APIs” on page 226.
TableImportAPI	Loads geographic zone data from the staging table into the operational table. This tool is supported in PolicyCenter for zone data only, not for other data such as policy data or administrative data. The methods of this web service are available only when the server run level is maintenance. See “Zone Import” on page 543.
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 Import” on page 543.
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 68.

Publishing Web Services

You can write web service APIs in Gosu and access them from remote systems using the standard web services protocol *SOAP*. The SOAP protocol defines request/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 or requesting data from another application across a network. PolicyCenter publishes its own built-in web service APIs that you can use.

This topic includes:

- “Web Service Publishing Overview” on page 42
- “Publishing and Configuring a Web Service” on page 46
- “Testing Web Services with Local WSDL” on page 50
- “Generating WSDL” on page 52
- “Adding Advanced Security Layers to a Web Service” on page 55
- “Web Services Authentication Plugin” on page 59
- “Checking for Duplicate External Transaction IDs” on page 61
- “Request or Response XML Structural Transformations” on page 61
- “Reference Additional Schemas in Your Published WSDL” on page 62
- “Validate Requests Using Additional Schemas as Parse Options” on page 62
- “Invocation Handlers for Implementing Preexisting WSDL” on page 63
- “Locale Support” on page 66
- “Setting Response Serialization Options, Including Encodings” on page 67
- “Exposing Typelists and Enumerations as String Values” on page 67
- “Transforming a Generated Schema” on page 68
- “Login Authentication Confirmation” on page 68
- “Stateful Session Affinity Using Cookies” on page 69
- “Calling a PolicyCenter Web Service from Java” on page 69

See also

- For information about the WS-I standard and its Document Literal encoding, see “Calling Web Services from Gosu” on page 75.

Web Service Publishing Overview

Web services define request-and-response APIs that let you call an API on a remote computer, or even the current computer, using an abstracted well-defined interface. A data format called the Web Service Description Language (WSDL) describes available web services that other systems can call using the SOAP protocol. Many languages or third-party packages provide bindings implementations of WSDL and SOAP, including Gosu (built into PolicyCenter), Java, Perl, and other languages.

Gosu natively supports web services in two different ways:

- **Publish your Gosu code as new web service APIs** – Write Gosu code that external systems call as a web service using the SOAP protocol. Simply add a single line of code before the definition of a Gosu class.
- **Call web service APIs that external applications publish from your Gosu code** – Write Gosu code that imports and calls web service APIs published by external systems. Gosu parses the 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 with a natural Gosu syntax. For more information, see “Calling Web Services from Gosu” on page 75.

Designing Your Web Services

In both cases PolicyCenter converts the server’s local Gosu objects to and from the flattened text-based format required by the SOAP protocol. This process is *serialization* and *deserialization*.

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

Guidewire provides built-in web service APIs for common general tasks and tasks for business entities of PolicyCenter. For a full list, see “Reference of All Built-in Web Services” on page 39. However, 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 elegantly 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, trigger actions, or to change data within the PolicyCenter database.

Publishing a web service may be as simple as added one special line of code called an annotation immediately before your Gosu class. Additional customizations may require extra annotations. See “Publishing and Configuring a Web Service” on page 46 for details.

There are special additional tasks or design decisions that affect how you write your web services, for example:

- **Create custom structures to send only the subset of data you need** – For large Guidewire business data objects (entities), most integration points only need to transfer a subset of the properties and object graph. Do not pass large object graphs, and be aware of any objects that might be very large in your real-world deployed production system. In such cases, you must design your web services to pass your own objects containing

only your necessary properties for that integration point, rather than pass the entire entity. For example, if an integration point only needs a record's main contact name and phone number, create a shell object containing only those properties and the standard public ID property. For details, see “Publishing and Configuring a Web Service” on page 46.

Web services published by Guidewire applications are forbidden to use a Guidewire entity type as an argument to a method or a return type. Instead, create other kinds of objects that store the important data for that integration point.

IMPORTANT WS-I web services cannot have arguments or return types that directly or indirectly access Guidewire entity types.

For example, you could change argument and return types to the following:

- **Gosu class instances** – Write Gosu classes that include the important properties for that integration point. For example, a check printing service might only need the amount and the mailing address, but not any associated metadata or notes about the check. To work with web services, you must add the `final` keyword on the class, and also add the `@WsiExportable` annotation on the class. For example:

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

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

- **XML types created with the Guidewire XML modeler** – Use the Guidewire XML modeler tool (also called the *GX modeler*) to generate a model that contains only the desired subset of properties for each entity for each integration point. Then you can import or export XML data using that GX modeler as needed. See “The Guidewire XML (GX) Modeler” on page 306 in the *Gosu Reference Guide*.
- **Be careful of big objects** – If your data set is particularly large, it may be too big to pass over the SOAP protocol in one request. You may need to refactor your code to accommodate smaller requests. If you try to pass too much data over the SOAP protocol in either direction, there can be memory problems that you must avoid in production systems.
- **Design your web service to be testable** – For an example of testing a web service, see “Testing Web Services with Local WSDL” on page 50.
- **Learn bundle and transaction APIs** – To change and commit entity data in the database, learn the bundle APIs. You do not need to use bundle APIs for SOAP APIs that simply get and return unchanged data. See “Committing Entity Data to the Database Using a Bundle” on page 44.

Guidewire provides a Java language binding for published web services. Using these bindings, you can call the published web services from Java program as easy as making a local method invocation. You can use other programming languages if it has a SOAP implementation and can access the web service over the Internet.

Because Java is the typical language for web service client code, this topic usually uses Java syntax and terminology to demonstrate APIs. In some cases, this topic uses Gosu to demonstrate examples as it relates to publishing or testing web services.

If you write APIs to integrate two or more Guidewire applications, you probably will write your web service code in Gosu using its native SOAP syntax rather than Java. See “Calling Web Services from Gosu” on page 75.

WARNING Avoid calling locally-hosted SOAP APIs from within a plugin or the rules engine in production systems. Be careful about any SOAP calls to the same server.

Committing Entity Data to the Database Using a Bundle

There is no default writable bundle for web services. Bundles are the way that Guidewire applications track changes to entity data. You must define your own new writable bundle if your web service must change any entity data.

To create a new writable bundle, use the `runWithNewBundle` API. See “Running Code in an Entirely New Bundle” on page 347 in the *Gosu Reference Guide*.

Web services that only get entity data (never modify entity data) never need to create a new bundle.

Serializable Gosu Classes Must Be Final and Exportable

Web services can contain arguments and return values that contain or reference regular instances of Gosu classes, sometimes called Plain Old Gosu Objects (POGOs).

However, the Gosu class must have the following two special qualities:

- The class must be declared as `final`, which as a consequence means it has no subclasses.
- The class must have the annotation `@WsIExportable` or Gosu does not publish the service.

Web Service Publishing Quick Reference

The following table summarizes important qualities of WS-I web services.

Feature	WS-I web service behavior
Basic publishing of a WS-I web service	Add the annotation <code>@WsIWebService</code> to the implementation class. This annotation supports one optional argument for the web service namespace. See “Declaring the Namespace for a Web Service” on page 46. If you do not declare the namespace, Gosu uses the default namespace <code>http://example.com</code> .
Does PolicyCenter automatically generate WSDL files from a running server?	Yes. See “Getting WSDL from a Running Server” on page 52.
Does PolicyCenter automatically generate WSDL files locally if you regenerate the SOAP API files?	Yes. See “Generating WSDL” on page 52.
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 69.
Can serialize Gosu class instances, sometimes called POGOs: Plain Old Gosu Objects?	Yes, however the Gosu class must have two special qualities. See “Serializable Gosu Classes Must Be Final and Exportable” on page 44.
Can web services serialize or deserialize Guidewire entity instances?	No. To transfer entity data, create data transfer objects (DTOs) as that contain only the data you need. DTO objects can be either Gosu class instances or XML objects. See later in this table for “Can serialize Gosu class instances”.
Can serialize a XSD-based type	Yes, using the <code>XmlElement</code> APIs.
Can you use the Guidewire XML Modeler tool (GX Model) to generate XML types that can be WS-I arguments or return types?	Yes
Can it serialize a Java object as an argument type or return type?	No
Automatically throw <code>SOAPException</code> exceptions?	No. Declare the actual exceptions you want thrown. In general this requirement reduces typical WSDL size.
Bundle handling for changing entity data?	A <i>bundle</i> is a container for entities that helps PolicyCenter track what changed in a database transaction. There is no default bundle for WS-I web services. See “Committing Entity Data to the Database Using a Bundle” on page 44.

Feature	WS-I web service behavior
Logging in to the server	Each WS-I 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 46.
Calling a local version of the web service from Gosu	Gosu creates types that use the original package name to avoid namespace collisions. API references in the package <code>wsi.local.ORIGINAL_PACKAGE_NAME</code> You must call the command line command to rebuild the local files: <code>PolicyCenter/bin/gwpc regen-wsi-local</code>

Web Service Publishing Annotation Reference

The following table lists annotations related to WS-I web service declaration and configuration:

Annotation	Description	For more information
Web service implementation class annotations		
<code>@WsiWebService</code>	Declare a class as implementing a WS-I web service	"Publishing and Configuring a Web Service" on page 46
<code>@WsiAvailability</code>	Set the minimum server run level for this service	"Specifying Minimum Run Level for a Web Service" on page 47
<code>@WsiPermissions</code>	Set the required permissions for this service	"Specifying Required Permissions for a Web Service" on page 47
<code>@WsiExposeEnumAsString</code>	Instead of exposing typelist types and enumerations as enumerations in the WSDL, you can expose them as <code>String</code> values.	"Exposing Typelists and Enumerations as String Values" on page 67
<code>@WsiWebMethod</code>	The <code>@WsiWebMethod</code> annotation can do two things: <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 <code>public</code> access. 	"Overriding a Web Service Method Name or Visibility" on page 48
<code>@WsiSerializationOptions</code>	Add serialization options for web service responses, for example supporting encodings other than UTF-8.	"Setting Response Serialization Options, Including Encodings" on page 67
<code>@WsiAdditionalSchemas</code>	Expose additional schemas to web service clients in the WSDL. Use this to provide references to schemas that might be required but are not automatically included.	"Reference Additional Schemas in Your Published WSDL" on page 62
<code>@WsiParseOptions</code>	Add validation of incoming requests using additional schemas in addition to the automatically generated schemas.	"Validate Requests Using Additional Schemas as Parse Options" on page 62

Annotation	Description	For more information
@WsiRequestTransform @WsiResponseTransform	Add transformations of incoming or outgoing data as a byte stream. Typically you would use this to add advanced layers of authentication or encryption. Contrast to the annotations in the next row, which operate on XML elements.	"Data Stream Transformations" on page 55
@WsiRequestXml1Transform @WsiResponseXml1Transform	Add transformations of incoming or outgoing data as XML elements. Use this 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 61
@WsiSchemaTransform	Transform the generated schema that PolicyCenter publishes.	"Transforming a Generated Schema" on page 68
@WsiInvocationHandler	Perform advanced implementation of a web service that conforms to externally-defined standard WSDL. This is for unusual situations only. This approach limits protection against some types of common errors.	"Invocation Handlers for Implementing Preexisting WSDL" on page 63
@WsiCheckDuplicateExternalTransaction	Detect duplicate operations from external systems that change data	"Checking for Duplicate External Transaction IDs" on page 61
Other annotations to support serialization		
@WsiExportable	Add this annotation on a Gosu class to indicate that it supports serialization with WS-I web services. You must additionally add the <code>final</code> keyword to the class to prevent subclasses.	"Serializable Gosu Classes Must Be Final and Exportable" on page 44

Publishing and Configuring a Web Service

To publish a WS-I web service, use the `@WsiWebService` annotation on a class. Gosu publishes that class automatically when the server runs.

For example:

```
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. This reduces the chance of namespace collisions.

Declaring the Namespace for a Web Service

Each web service is defined within a *namespace*, which is similar to how a Gosu class or Java class is 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 this 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. For example:

- "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.

For example:

```
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. For examples of how the Java `wsimport` tool uses the namespace, see "Calling a PolicyCenter Web Service from Java" on page 69.

Specifying 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.

For example:

```
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 our list of permissions, your web service does not require SOAPADMIN permission. If you pass an empty list of permissions, your web service does not require authentication at all.

The only supported permission types are permissions that are role-based (static), rather than data-based (requires an object). In the *Security Dictionary*, view the desired permission. If the permission is role-based, the page says after the permission name the word "(static)".

The following example removes 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

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 simply 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. For example:

```

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

```

Hiding Public Web Service Methods

By default, Gosu publishes one web service operation for each method marked with `public` availability. For `public` methods, you can exclude the method from the web service by adding the `@WsiWebMethod` annotation immediately before the declaration for the method on the implementation class. Pass the value `true` to exclude (suppress) publishing that public method. For example:

```

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

```

If you pass the value `false` instead of `true` to the `@WsiWebMethod` annotation, there is no different behavior compared to omitting the annotation. If you use this annotation on a non-public method, the exclusion behavior has no effect. It never forces a non-public method to be visible on the web service.

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. For example:

```

@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 properties on `WsiInvocationContext` objects and whether each property is writable.

Property	Description	Readable	Writable
Request Properties			
<code>HttpServletRequest</code>	A servlet request of type <code>HttpServletRequest</code>	Yes	No
<code>MtomEnabled</code>	<p>A boolean value that specifies whether to optionally support sending MTOM attachments to the WS-I web service client in any data returned from an API. The W3C Message Transmission Optimization Mechanism (MTOM) is a method of efficiently sending binary data to and from web services as attachments outside the normal response body.</p> <p>If <code>MtomEnabled</code> is <code>true</code>, PolicyCenter can send MTOM in results. Otherwise, MTOM is disabled. The default is <code>false</code>.</p> <p>This property does not affect MTOM data sent to a published web service. Incoming MTOM data is always supported.</p> <p>This property does not affect MTOM support where Gosu is the client to the web service request. See "MTOM Attachments with Gosu as Web Service Client" on page 91.</p>	Yes	Yes
<code>RequestHttpHeaders</code>	Request headers of type <code>HttpHeaders</code>	Yes	No
<code>RequestEnvelope</code>	Request envelope of type <code>Xmlelement</code>	Yes	No
<code>RequestSoapHeaders</code>	Request SOAP headers of type <code>Xmlelement</code>	Yes	No
Response Properties			
<code>ResponseHttpHeaders</code>	Response HTTP headers of type <code>HttpHeaders</code>	Yes	The property value is read-only, but you can modify the object it references
<code>ResponseSoapHeaders</code>	Response SOAP headers of type <code>List<Xmlelement></code>	Yes	The property value is read-only, but you can modify the object it references
Output serialization			
<code>XmlSerializationOptions</code>	<p>XML serialization options of type <code>XmlSerializationOptions</code>. For more information on XML serialization, including the properties of <code>XmlSerializationOptions</code> objects, see "Exporting XML Data" on page 279 in the <i>Gosu Reference Guide</i>.</p>	Yes	Yes

You can use these APIs 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. Then you can use code with the invocation context such as the following to get the incoming request header and add it to the response headers:

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

    // do your main work of your web service operation
}
```

Web Service Class Lifecycle and Request Local Scoped Variables

PolicyCenter does not instantiate the web service implementation class on server startup. PolicyCenter instantiates the web service implementation class upon 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 may share this object, you must be careful with use of static variables or instance variables. You must use concurrency APIs to ensure thread safety. For most Gosu coding, you can use the standard Gosu concurrency classes `RequestVar` and `SessionVar` in package `gw.api.web` package. However, these classes do not work in web service implementation classes.

Instead, for web service request-based data storage, use the concurrency class `gw.xml.ws.WsiRequestLocal`. If you use this for web service implementation class instance variable, the object exists only once. However, the object has `get` and `set` methods that manage the variable life cycle for each request in a thread-safe way. In this context, a request refers to invocation of one web service operation (one method of your web service).

To define a web request local scoped variable

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 using 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. For example:

```
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`.

Testing Web Services with Local WSDL

For testing purposes only, you can call WS-I web services published from the same PolicyCenter server. To call a WS-I 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.

WARNING Guidewire does not support calls to SOAP APIs published on the same server in a production system. If you think you need to do so, please contact Customer Support.

To regenerate the WSDL for all local web services in the class hierarchy, open a command prompt and type:

```
PolicyCenter/bin/gwpc regen-wsi-local
```

PolicyCenter generates the WSDL in the following location in the class hierarchy. The WSDL is in the `wsi.local` package, followed by fully qualified name of the web service:

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

To use the class, simply 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 `regen-wsi-local` tool generates the following WSDL file:

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

Call this web service locally with the syntax:

```
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 `regen-wsi-local` tool, it creates new files in the `configuration` module.

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

To reduce the chance of accidental use of `wsi.local.*` types, Gosu prevents using these types in method signatures of published WS-I 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 assure 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 assure the correctness of database transactions, test your web service to exercise all bundle-related code.

Generating WSDL

Getting WSDL from a Running Server

Typically, you get the WSDL for a WS-I 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 around risks callers of the web service using outdated WSDLs. 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 WS-I 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 `ws.eg.WebService.TestWsService` at the location on a server with web application name pc:

`http://localhost:PORTNUM/pc/ws/eg/webservice/TestWsService?WSDL`

Java includes a built-in command called `wsimport` that generates Java from the WSDL published on a running server. For more information, see “Calling a PolicyCenter Web Service from Java” on page 69

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:8580/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.

Generating WSDL On Disk

Typically, clients of a WS-I web service get the WSDL from the server where the web service runs to ensure that they use the current WSDL, not an outdated version. Also, you can get that WSDL from anywhere on your network. In a production system, your SOAP client code typically runs on computers that are separate from the computer where your SOAP server code runs.

However, there are special situations in which you might want to generate the WSDL file locally on the server. Locally generated WSDL files require a special annotation in your web service code and a special regeneration tool step.

Special Annotation to Generate WSDL On Disk

To generate WSDL for a WS-I web service locally on the server where the service runs, you must add the annotation `@WsiGenInToolkit` to the web service implementation class. Most WS-I web services included with PolicyCenter do not have the annotation. If you want to generate the WSDL locally for these web services, modify the implementation file in Studio to add the `@WsiGenInToolkit` annotation.

For example, the following web service implementation class has the annotation before the class declaration:

```

package example

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

@WsiWebService
@WsiGenInToolkit

class HelloWorldAPI {

    public function helloWorld() : String {
        return "Hello!"
    }
}

```

Command Line tool to Generate WSDL On Disk

To generate the WSDL for all the WS-I web services with the annotation `@WsiGenInToolkit`, at a command prompt type the command:

```
PolicyCenter/bin/gwpc regen-soap-api
```

Look for the locally generated WSDL files in the directory:

```
PolicyCenter/soap-api/wsi/wsdl
```

Generating SOAP APIs for Use with Unit Tests From Gosu

To support writing Gosu unit tests (JUnit tests), PolicyCenter can generate WSDL within the class file hierarchy for any published WS-I web services. This permits you to write unit tests that access the WSDL files over the SOAP protocol from Gosu. Other than for the purpose of unit tests, Guidewire does not support calls from a PolicyCenter server into the same server over the SOAP protocol.

See also

- “Testing Web Services with Local WSDL” on page 50

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.

Data Stream Transformations

Transformations on Data Streams

You can transform the data stream by processing it incrementally, byte by byte. For example, you can implement an encryption security layer by transforming the request data stream incrementally. Alternatively, you can transform the data stream by processing it as an entire unit at one time. For example, you must implement a digital signature authentication layer by transforming the entire request data stream at one time.

You can apply multiple types of transformations to the request data stream to add multiple security layers to your web service. The order of your transformations is important. For example, an encryption transformation followed by a digital signature transformation produces a different request data stream than a digital signature transformation followed by an encryption transformation.

If your desired transformation operates more naturally on XML elements and not a byte stream, instead consider using the APIs in “Request or Response XML Structural Transformations” on page 61

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.

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 (\wedge) to perform the bitwise changes on each byte. The XOR logical operator is a *symmetrical* operation. If you apply 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  $\wedge$  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 scenario however, 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 57
- “Bitwise Exclusive OR (\wedge)” on page 71 in the *Gosu Reference Guide*

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, assure the remaining layers preserve the correct order of substeps in the transformation blocks.

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 WS-I 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  
    }  
  
}
```

Some things to notice about the preceding web service implementation class:

- The web service itself simply adds two numbers, but the service has a request and response transformation.
- The request transformation removes and confirms the cryptographic layer on the request, including the digital signature and encryption. The request transformation calls `DemoCrypto.removeCrypto(inputStream)`.
- The response transformation adds the cryptographic layer on the response. The response transformation calls `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.

See also

- “Testing Web Services with Local WSDL” on page 50

Web Services Authentication Plugin

To handle the name/password authentication for a user connecting to WS-I web services, PolicyCenter delegates this job 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 WS-I web service connections only.

The Default Web Services Authentication Plugin Implementation

To authenticate web service requests, it is common to set up non-human Guidewire application users through regular PolicyCenter user administration. These non-human users never use their usernames and passwords to sign into the ClaimCenter user interface. Instead, an external system passes the username and password in the request headers of its WS-I web service requests to authenticate.

The default configuration of PolicyCenter has a registered built-in implementation of WS-I web service authentication plugin. The class is called `gw.plugin.security.DefaultWebservicesAuthenticationPlugin`. In the default configuration of PolicyCenter, this class does two things:

1. Looks at HTTP request headers for WS-I authentication information.
2. Performs authentication against the local PolicyCenter users in the database. This class calls the registered implementation of the `AuthenticationServicePlugin` plugin interface.

IMPORTANT If you write your own implementation of the `AuthenticationServicePlugin` plugin interface, be aware of this interaction with WS-I web service authentication. For example, you might want LDAP authentication for most users, but for web service authentication 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 and instead authenticate with the local application administrative data. To do this in your `AuthenticationServicePlugin` implementation, use the code:

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

For details of authentication handlers and the `AuthenticationServicePlugin` plugin interface, see “User Authentication Service Plugin” on page 185.

Writing an Implementation of the Web Services Authentication Plugin

Most customers do not need to write a new implementation of the `WebservicesAuthenticationPlugin` plugin interface. Typical changes to WS-I authentication logic are instead in your `AuthenticationServicePlugin` plugin implementation. See related discussion in “The Default Web Services Authentication Plugin Implementation” on page 60.

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 plugin method that your implementation must implement, called `authenticate`. The `authenticate` method takes one parameter of type `gw.plugin.security.WebservicesAuthenticationContext`. The object passed to your `authenticate` method contains authentication information, such as the name and password.

Properties on a Web Services Authentication Context

Important properties on a `WebservicesAuthenticationContext` include:

- `HttpHeaders` – HTTP headers of type `gw.xml.ws.HttpHeaders`. This includes a list of header names
- `HttpServletRequest` – the HTTP servlet request object, as the standard Java object `javax.servlet.http.HttpServletRequest`.
- `RequestSOAPHeaders` – The request SOAP headers, as an XML element (`XmlElement`)

Values to Return from Your Default Web Services Authentication Plugin

The value that your implementation returns from its `authenticate` method depend on whether authentication succeeds:

- If authentication succeeds, return the relevant `User` object from the PolicyCenter database.
- If you cannot attempt to authenticate for some reason, such as network problems, return `null`.
- If authentication fails for other reasons, throw an exception of type `WsiAuthenticationException`.

Checking for Duplicate External Transaction IDs

To detect duplicate operations from external systems that change data, add the `@WsiCheckDuplicateExternalTransaction` annotation to your WS-I 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 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.

If your WS-I client code is written in Gosu, to set the SOAP header see “Setting Guidewire Transaction IDs” on page 85.

If you apply this feature to an operation (or to the whole class), and the SOAP header `<transaction_id>` is missing, PolicyCenter throws an exception.

Request or Response XML Structural Transformations

For advanced layers of security, you probably want to use transformations that use the byte stream. See “Data Stream Transformations” on page 55.

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.Envelope`

Reference 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 to do them. Use this 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 accepts 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 web service client code can access them from the WSDL for the service.

The annotation takes one argument of the type `List<XmlSchemaAccess>`, which means a list of schema access objects. To get a reference to a schema access object, first put a 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, simply 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:`

```
@WsiAdditionalSchemas({ gw.xml.ws.wsimyschema.util.SchemaAccess })
```

See also

- “Reference of XSD Properties and Types” on page 286 in the *Gosu Reference Guide*
- “Referencing Additional Schemas During Parsing” on page 283 in the *Gosu Reference Guide*
- “Transforming a Generated Schema” on page 68

Validate Requests Using Additional Schemas as Parse Options

You can validate incoming requests 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 reference. 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 } })
```

See also

- “Introduction to the XML Element in Gosu” on page 275 in the *Gosu Reference Guide*
- “Schema Access Type” on page 305 in the *Gosu Reference Guide*
- “Reference Additional Schemas in Your Published WSDL” on page 62

Invocation Handlers for Implementing Preexisting WSDL

The implementation of a Gosu WS-I web service is a Gosu class. Typically, each WS-I operation corresponds directly to one method on the Gosu class that implements the web service. Gosu automatically determines and generates the output WSDL for that operation. For any method arguments and return types, Gosu uses the class definition and the method signatures to determine what to export in the WSDL.

However, if necessary you can use argument types and return values defined in a separate WSDL. This might be necessary for example if you are required to implement a preexisting service definition. Perhaps ten different systems implement this service, and only one is defined by Gosu. By using a standardized WSDL, some organization can ensure that all types sent and received conform to a standard WSDL definition for the service.

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. Generally speaking, using this approach makes your code harder to read code and 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 various 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 WS-I parse options annotation `@WsiParseOptions`. Pass the entire WSDL as the schema as described in that topic.

See also

- “Introduction to the XML Element in Gosu” on page 275 in the *Gosu Reference Guide*
- “XSD-based Properties and Types” on page 285 in the *Gosu Reference Guide*
- “Validate Requests Using Additional Schemas as Parse Options” on page 62

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. See “Introduction to the XML Element in Gosu” on page 275 in the *Gosu Reference Guide*.

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.

See also

- “Web Service Invocation Context” on page 48

Invocation Handler Responsibilities

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

To re-enable bypassed WS-I features from an invocation handler

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. For example, the method signature:

```
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 desired 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. See “Checking for Duplicate External Transaction IDs” on page 61.
- 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 WS-I 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 WS-I 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 desired method from the invocation handler.

Assuming that you want to check the method-specific annotations for run level or permissions, one potential 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 WS-I 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 is the only supported use of a `WsiInvocationContext` object's `preExecute` method. Any use other than calling it exactly once from within an invocation handler `invoke` method is unsupported.

Locale Support

By default, WS-I web services use the default server locale.

WS-I web service clients can override this behavior and set a locale to use while processing this web service request. To set the locale, the client can add a SOAP header element type `<gwsoap:locale>` with namespace "`http://guidewire.com/ws/soapheaders`", with the element containing the locale code.

For example, the following SOAP envelope contains a SOAP header that sets the `fr_FR` locale for French language in France:

```
<soap12:Envelope xmlns:soap12="http://www.w3.org/2003/05/soap-envelope">
<soap12:Header>
    <gwsoap:locale xmlns:gwsoap="http://guidewire.com/ws/soapheaders">fr_FR</gwsoap:locale>
</soap12:Header>
<soap12:Body>
    <getPaymentInstrumentsFor xmlns="http://example.com/gw/webservice/pc/pc700/PaymentAPI">
        <accountNumber>123</accountNumber>
    </getPaymentInstrumentsFor>
</soap12:Body>
</soap12:Envelope>
```

In this case, the web service would set the `fr_FR` locale before processing this web service request.

On a related topic, you can quickly configure the locale in a web service client in Gosu, such as from another Guidewire application. See “Setting Locale in a Guidewire Application” on page 88. That configuration information sets the SOAP header mentioned in this section.

Setting Response Serialization Options, Including Encodings

You can customize how PolicyCenter serializes the XML in the web service response to the client.

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.

Note: 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.

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" ) } )
```

For more information about other XML serialization options, such as indent levels, pretty printing, line separators, and element sorting, see “Exporting XML Data” on page 279 in the *Gosu Reference Guide*.

If the web service client is a Guidewire product, you can configure the character encoding for the request in addition to the server response. See “Setting XML Serialization Options” on page 87.

Exposing Typelists and Enumerations as 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.

For example:

```
<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.
- Changing enumeration values even slightly can cause 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 WS-I web service client code might be middleware that simply 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.

You can expose multiple types as `String` values by repeating the annotation multiple types on separate lines, each providing a different 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 )
```

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.xmlschema.Schema`. There may be additional schemas in the WSDL that are unrepresented by this parameter but are accessible through the WSDL parameter.

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

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

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

Login Authentication Confirmation

In typical cases, WS-I 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 WS-I web service as a precondition for login authentication. In effect, WS-I 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. 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 WS-I 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, you must call the matching `logout` method to clear the session, passing the session ID as an argument. If you were merely trying to confirm authentication, 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 fail to call `logout`, all application servers are configured to time out the session eventually.

Stateful Session Affinity Using Cookies

By default, WS-I web services do not ask 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*.

The WS-I 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 customer portal integration. Repeated page requests by the same user (assuming they successfully reused the same 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`" (with no quotes) to the WS-I 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.ab800.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 a PolicyCenter Web Service from Java

For WS-I 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:

- Java 6 (Java 1.6) includes a built-in utility to generate Java-compatible stubs using the `wsimport` tool.
- The CXF open source tool.
- The Axis2 open source tool.

Calling Web Services using Java 1.6 and wsimport

Java 6 (Java 1.6) includes a built-in utility that generates Java-compatible stubs from WSDL. This tool is called `wsimport` tool. This documentation uses this built-in Java tool and its output to demonstrate creating client connections to the PolicyCenter web services.

You can get the WSDL for the WS-I web service from one of two sources:

- “Getting WSDL from a Running Server” on page 52
- “Generating WSDL On Disk” on page 54

The `wsimport` tool supports getting WSDL over the Internet published directly from a running application. This is the approach that this documentation uses to demonstrate how to use `wsimport`.

Note: The `wsimport` tool also supports using local WSDL files. Refer to the `wsimport` tool documentation for details.

To generate Java classes that make SOAP client calls to a SOAP API

1. Launch the server that publishes the WS-I 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. Decide the name of the subdirectory of the current directory where you want to place the Java source files. For example, you might choose the folder name `src`. Assure the subdirectory already exists before you run the `wsimport` tool in the next step. If the directory does not already exist, create the directory now. This topic calls this the `SUBDIRECTORY_NAME` directory.
5. Type the following command:

```
wsimport WSDL_LOCATION_URL -s SUBDIRECTORY_NAME
```

For `WSDL_LOCATION_URL`, type the HTTP path to the WSDL. See “Getting WSDL from a Running Server” on page 52.

For example:

```
wsimport http://localhost:PORTNUMBER/pc/ws/example/HelloWorldAPI?WSDL -s src
```

IMPORTANT You must assure the `SUBDIRECTORY_NAME` directory already exists before running the command. If the directory does not already exist, the `wsimport` action fails with the error “`directory not found`”.

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.
 - The `.java` files are in the `SUBDIRECTORY_NAME` subdirectory. To use these, 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 represent 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`”. For more details, see “Declaring the Namespace for a Web Service” on page 46.

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 fully qualified name is `com.mycompany.HelloWorld` and the namespace is the default "`http://example.com`". Suppose 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`

- 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`. In other words, the location is:
`CURRENT_DIRECTORY/SUBDIRECTORY_NAME/REVERSED_NAMESPACE/FULLY_QUALIFIED_NAME`
- Continuing our example, the `wsimport` tool generates the `.class` files at the following location:
`CURRENT_DIRECTORY/com/example/com/mycompany/HelloWorld.class`
- To use the Java class files, add this directory to your Java project's class path, or copy the files to your Java project's `src` folder.

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.

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

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

For example, for the API interface name `HelloWorldAPI`, the Java code looks like:

```
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 47. 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 WS-I web service
        String res = port.helloWorld();

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

See also

- “Adding HTTP Basic Authentication in Java” on page 71
- “Adding SOAP Header Authentication in Java” on page 72

Adding HTTP Basic Authentication in Java

In previous topics the examples showed how to connect to a service without authentication. The following code shows how to add HTTP Basic authentication to your WS-I client request. HTTP Basic authentication is the easiest type of authentication to code to connect to a PolicyCenter WS-I web service.

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

```

import com.example.example.helloworldapi.HelloWorldAPIPortType;
import com.sun.xml.internal.ws.api.message.Headers;
import javax.xml.ws.BindingProvider;
import org.w3c.dom.Document;
import org.w3c.dom.Element;

import javax.xml.namespace.QName;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.ws.BindingProvider;
import java.util.Map;

public class WsiTest02 {

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

        // get a reference to the SOAP 1.1 port for this web service
        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 WS-I web services using custom SOAP headers.

For Guidewire applications, the structure of the required SOAP header is:

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

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

The Guidewire authentication XML element looks like the following:

```

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

```

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

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

```

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

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

public class WsiTest01 {

```

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

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

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

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

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

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

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

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

    // Add our 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);
}

}
```


Calling Web Services from Gosu

Gosu code can import web services (SOAP APIs) from external systems and call these services as a SOAP client (an API consumer). The Gosu language handles all aspects of object serialization, object deserialization, basic authentication, and SOAP fault handling.

This topic includes:

- “Consuming Web Service Overview” on page 75
- “Adding Configuration Options” on page 82
- “One-Way Methods” on page 89
- “Asynchronous Methods” on page 90
- “MTOM Attachments with Gosu as Web Service Client” on page 91

Consuming Web Service Overview

Gosu supports calling WS-I compliant web services. 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. See “Asynchronous Methods” on page 90.
- Support one-way web service methods. See “One-Way Methods” on page 89.
- Separately encrypt requests and responses. See “Adding Configuration Options” on page 82.
- Process attachments that use the multi-step MTOM protocol. See “MTOM Attachments with Gosu as Web Service Client” on page 91.
- Sign incoming responses with digital signatures. See “Implementing Advanced Web Service Security with WSS4J” on page 88.

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 an industry-standard 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.

Loading WSDL Locally Using Studio Web Service Collections

The recommended way of consuming WS-I web services is to use a web service collection in Studio. A web service collection encapsulates one or more web service endpoints, and any WSDL or XSD files they reference. If you ever want to refresh the downloaded WSDL or XSD files in the collection, simply navigate to the web service collection editor in Studio and click **Fetch Updates**.

IMPORTANT To create a web service collection in Studio, see “Using the Web Service Editor” on page 138 in the *Configuration Guide* for details.

For adding configuration options to the connection, such as authentication and security settings, there are multiple approaches. See “Adding Configuration Options” on page 82.

Web service collection (.wsc) files encapsulate the set of resources necessary to connect to a web service on an external system. If you view a web service collection in Studio and click the **Fetch Updates** button, Studio retrieves WSDL and XSD files from the servers that publish those web services. You can trigger the **Fetch Updates** process from a command line tool called `regen-from-wsc`:

- First, be sure that your `suite-config.xml` file correctly refers to your server names and port numbers for your Guidewire applications. See “Suite Configuration File Sets URLs to Guidewire Applications” on page 84.
- Next, from a command prompt in the `PolicyCenter/bin` directory, type the following:
`gwpc regen-from-wsc`

The tool refreshes all of your .wsc files in your Studio configuration.

Loading WSDL Directly into the File System

To consume an external web service, you must load the WSDL and XML schema files (XSDs) for the web service. You must fetch copies of WSDL files, as well as related WSDL and XSD files, from the web services server. Fetch the copies into an appropriate place 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.

To automatically fetch WSDLs and XSDs from a web service server

1. Within Studio, navigate within the `Classes → wsi` hierarchy to a package in which you want to store your collection of WSDL and XSD files. Guidewire recommends that you place your web service collection in the `Resources → Configuration → Classes → wsi → remote` hierarchy.
2. Right-click and choose **New → Webservice Collection**. Studio prompts you for a name for the web service collection. Enter a name for the web service collection and click **OK**.

3. Click **Add Resource...**
4. Enter the URL of the WSDL for the external web service. This is also called the *web service endpoint URL*. For example, enter *WebServiceURL/WebServiceName.wso?wsdl*. Studio indicates whether the URL is valid. You cannot proceed until you enter a valid URL. After determining that the URL is valid, click **OK**.
5. Studio indicates that you modified the list of resource URLs and offers to fetch updated resources. Click **Yes**. You can click **Fetch Updates** at any time to refresh the WSDL from the web service server.
6. Studio retrieves the WSDL for that service. You see the resource URL in the editor's **Resources** pane.
7. Click **Fetched Resources** to view the WSDL and its associated resources.

Web Service Collections in the File System

When you fetch WSDL resources, Studio retrieves the WSDL from the web service and stores it locally at the path location in Studio where you defined your web service collection. Studio downloads the WSDL and any related XSDs into a subdirectory of that location. The subdirectory has the same name as your web service collection.

The path of the WSDL is *PACKAGE_NAME/web_service_name.wsdl*. Studio creates the value for *web_service_name* from the last part of the web service endpoint URL, not from the WSDL. For example, the URL is *MY_URL/info.wsdl* or *MY_URL/info.wso?wsdl*. Gosu creates all the types for the web service in the namespace *PACKAGE.info*.

Sample Code to Manually Fetch WSDLs and XSDs from a Web Service Server

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 --  
Wsd12Gosu.fetch(new URL(urlStr), new File(loc))
```

The first long string (*urlStr*) is the URL to the web service WSDL. The second long string (*loc*) is 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.

Security and Authentication

The WS-I basic profile requires support for some types of security standards for web services, such as encryption and digital signatures (cryptographically signed messages). See “Adding Configuration Options” on page 82.

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). See “Asynchronous Methods” on page 90.
- One-way methods, which indicate a method defined to have no SOAP response at all. See “One-Way Methods” on page 89.

How Does Gosu Process WSDL?

As mentioned before, Studio adds a WSDL file to the class hierarchy automatically for each registered web service 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 namespace:

```
ws.myservice.*
```

Suppose you add a WSDL file directly to your class hierarchy called `MyService.wsdl` in the package `example.p1.gs.wsic`. Gosu creates all the types for your web service in the namespace:

```
example.p1.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.

For details, see “Normalization of Gosu Generated XSD-based Names” on page 288 in the *Gosu Reference Guide*.

The structure of a WSDL comprises the following:

- 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 looks like the following:

```
<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`. Let us assume for this first example 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 using code such as:

```
// 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 .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 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 looks 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 using code similar to the following:

```
// 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 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.

Note: If you use Guidewire Studio, you do not need to manipulate the WSDL file manually. Studio automatically gets 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. Internally, 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.

Be sure to read the warnings in the section “Web Service API Objects Are Not Thread Safe” on page 79.

Web Service API Objects Are Not Thread Safe

To create a WS-I API object, use a `new` expression to instantiate the right fully-qualified type:

```
var service = new .myservice.SayHello()
```

Be warned that the WS-I API object is not thread-safe in all cases.

WARNING 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. See “Setting Guidewire Transaction IDs” on page 85.

It is safest to follow these 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.

Learning 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.

Other 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. See “XSD-based Properties and Types” on page 285 in the *Gosu Reference Guide*.
- 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 details, see “Automatic Creation of Intermediary Elements” on page 299 in the *Gosu Reference Guide*.
- 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 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()
```

See also

- “Automatic Insertion into Lists” on page 290 in the *Gosu Reference Guide*
- “XSD-based Properties and Types” on page 285 in the *Gosu Reference Guide*
- “Gosu and XML” on page 271 in the *Gosu Reference Guide*

What Gosu Creates from Your WSDL

Within the namespace of the package of the WSDL, Gosu creates some new types.

For each service in the web service, Gosu creates a service by name. For example, if the external service has a service called `GeocodeService` and the WSDL is in the package `examples.gosu.wsdl`, then the service has the fully-qualified type `examples.gosu.wsdl.GeocodeService`. Create a new instance of this type, and you then you can call methods on it for each method.

For each operation in the web service, generally speaking Gosu creates two local methods:

- One method with the method name in its natural form, for example suppose a method is called `doAction`
- One method with the method name with the `async_` prefix, for example `async_doAction`. This version of the method handles asynchronous API calls. For details, see “Asynchronous Methods” on page 90.

Special Behavior For Multiple Ports

Gosu automatically processes ports from the WSDL identified as either SOAP 1.1 or SOAP 1.2. If both are available for any service, Gosu ignores the SOAP 1.1 ports. In some cases, the WSDL might define more than one available port (such as two SOAP 1.2 ports with different names).

For example, suppose you add a WSDL file to your class hierarchy called `MyService.wsdl` in the package `example.pl.gs.wsic`. Gosu chooses a default port to use and creates types for the web service at the following path:

```
ROOT_PACKAGE.WSDL_NAME_NORMALIZED.NORMALIZED_SERVICE_NAME
```

The `NORMALIZED_SERVICE_NAME` name of the package is the name of the service as defined by the WSDL, with capitalization and conflict resolution as necessary. For example, if there are two services in the WSDL named `Report` and `Echo`, then the API types are in the 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.

A Real Example: Weather

There is a public free web service that provides the weather. You can get the WSDL for this web service at the URL <http://wsf.cdyne.com/WeatherWS/Weather.asmx?wsdl>. This web service does not require authentication or encryption.

In Studio, navigate to the package `ws.weather` in the `Classes` hierarchy. Choose `New → Web Service Collection`. Enter the previously mentioned URL in the top field.

IMPORTANT There are other fields in the Studio user interface for configuring the web service collection in Studio. Refer to “Using the Web Service Editor” on page 138 in the *Configuration Guide* for details.

The following Gosu code gets the weather in San Francisco:

```
var ws = new ws.weather.Weather()  
var r = ws.GetCityWeatherByZIP(94114)  
print( r.Description )
```

Depending on the weather, your result might be something like:

```
Mostly Sunny
```

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. See “Introduction to the XML Element in Gosu” on page 275 in the *Gosu Reference Guide*.

Adding Configuration Options

If a web service does not need encryption, authentication, or digital signatures, you can just 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.wsdl.myservice.SayHello()

// call a method on the service
api.helloWorld()
```

If you need to add encryption, authentication, or digital signatures, there are two approaches

- **Configuration objects** – 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. For details, see “Directly Modifying the WSDL Configuration Object for a Service” on page 82.
- **Configuration providers** – Add one or more WS-I web service configuration providers in the Studio web service collection `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. 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. This 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.

[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, simply use a straightforward syntax to set authentication and security parameters. The following subtopics describe `WsdlConfig` object properties that you can set on the WSDL configuration object.

Note: 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. See also “Automatic Creation of Intermediary Elements” on page 299 in the *Gosu Reference Guide*.

Adding Configuration Provider Classes (To Centralize Your WSDL Configuration)

You can add one or more WS-I web service configuration providers in the Studio web service collection **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.

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 with signature:

```
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. For more details, see “[Directly Modifying the WSDL Configuration Object for a Service](#)” on page 82.

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:

- 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**.

WARNING 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.

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.IWsWebServiceConfigurationProvider

class ABConfigurationProvider implements IWsWebServiceConfigurationProvider {

    override function configure(serviceName : QName, portName : QName, config : WsdlConfig) {
        config.Guidewire.Authentication.Username = "su"
        config.Guidewire.Authentication.Password = "gwg"
    }
}
```

```
}
```

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 bother with adding these options with each use of the service. (For more information about Guidewire authentication, see “Guidewire Authentication” on page 84.)

HTTP Authentication

To add simple HTTP authentication to an API request, use the basic HTTP 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.

For example:

```
// Get a reference to the service in the package namespace of the WSDL.  
var service = new example.gosu.wsi.myservice.SayHello()  
  
service.Config.Http.Authentication.Basic.Username = "jms"  
service.Config.Http.Authentication.Basic.Password = "b5"  
  
// Call a method on the service.  
service.helloWorld()
```

Guidewire Authentication

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.

For example:

```
// 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()
```

Suite Configuration File Sets URLs to Guidewire Applications

PolicyCenter uses a single configuration file for configuring all web service URLs to other Guidewire InsuranceSuite applications. For Guidewire InsuranceSuite integrations that use WS-I web services, always use this suite configuration file to set the URLs. The suite configuration file is called `suite-config.xml`. In Studio, find it under **Configuration** → **config** → **suite**, then click `suite-config.xml`. Alternatively, type Shift-Ctrl+N and type the file name.

By default, all subelements are commented out. If you have multiple Guidewire products on one 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.

You can use the system environment setting `env` to trigger different values within the file. Each `<product>` element supports the `env` attribute to specifies the element is enabled only for that environment setting. See “Defining the Application Server Environment” on page 14 in the *System Administration Guide*.

The following example shows a URL with different values for development and production `env` settings:

```
<suite-config xmlns="http://guidewire.com/config/suite/suite-config">
  <product name="ab" url="http://localhost:8080/cc" env="development"/>
  <product name="ab" url="http://ProductionClaimCenterServer:8080/cc" env="production"/>
</suite-config>
```

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 this is found, PolicyCenter ignores any other ports on the WSDL for this service.
2. If so, it looks for shortcuts of the syntax `${PRODUCT_NAME_SHORTCUT}`. For example: `${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 `${PRODUCT_NAME_SHORTCUT}`. 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`. For example:

```
<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>
```

IMPORTANT When integrating with Guidewire InsuranceSuite applications, always use the `suite-config.xml` file to centrally configure domain names and port numbers of all applications.

Accessing the Suite Configuration File Using APIs

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.

For example:

```
uses gw.api.suite.GuidewireSuiteUtil
var x = GuidewireSuiteUtil.getProductInfo("cc")
print(x.Url)
```

For the earlier `suite-config.xml` example file, this code prints:

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

Setting Guidewire Transaction IDs

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. For implementation details, see “Checking for Duplicate External Transaction IDs” on page 61.

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.

IMPORTANT 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.

For example:

```
uses gw.xsd.guidewire.soapheaders.TransactionID
uses gw.xml.ws.WsdlConfig
uses java.util.Date
uses wsi.local.gw.services.wsidbupdateservice.faults.DBAlreadyExecutedException

function callMyWebService {

    // Get a reference to the service in the package namespace of the WSDL.
    var service = new example.gosu.wsi.myservice.SayHello()

    service.Config.Guidewire.Authentication.Username = "su"
    service.Config.Guidewire.Authentication.Password = "gw"

    // create a transaction ID that has an external system prefix and then a guaranteed unique ID
    // If you are using Guidewire messaging, you may want to use the Message.ID property in your ID.
    transactionIDString = "MyExtSys1:" + getUniqueTransactionID() // somehow 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() // somehow 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

To set the timeout value (in milliseconds), set the `CallTimeout` property on the `WsdlConfig` object for that API reference.

For example:

```
// get a reference to the service in the package namespace of the WSDL
var service = new example.gosu.wsi.myservice.SayHello()

service.Config.CallTimeout = 30000 // 30 seconds

// call a method on the service
service.helloWorld()
```

Custom SOAP Headers

SOAP HTTP headers are essentially XML elements attached to the SOAP envelope for the web service request or its response. Your code might need to send additional SOAP headers to the external system, such as custom headers for authentication or digital signatures. You also might want to read additional SOAP headers on the response from the external system.

To add SOAP HTTP headers to a request that you initiate, first construct an XML element using the Gosu XML APIs (`XmLElement`). Next, add that `XmLElement` object to the list in the location `api.Config.RequestSoapHeaders`. That property contains a list of `XmLElement` objects, which in generics notation is the type `java.util.ArrayList<XmLElement>`.

To read (get) SOAP HTTP headers from a response, it only works if you use the asynchronous SOAP request APIs described in “Asynchronous Methods” on page 90. 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.

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. It takes a `java.net.URI` object for the URL.

For example:

```
// get a reference to the service in the package namespace of the WSDL
var service = new example.gosu.wsi.myservice.SayHello()

service.Config.ServerOverrideUrl = new URI("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.

For full information about XML serialization options, such as encoding, indent levels, pretty printing, line separators, and element sorting, see “Exporting XML Data” on page 279 in the *Gosu Reference Guide*.

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.

For example, to change 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.wsi.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. See “Setting Response Serialization Options, Including Encodings” on page 67.

Setting Locale in a Guidewire Application

WS-I web services published on a Guidewire application support setting a specific international locale to use (to override) while processing this web services request.

For example:

```
// get a reference to the service in the package namespace of the WSDL
var service = new example.gosu.ws.i.myservice.SayHello()

// set the locale to the French language in the France region
service.Config.Guidewire.Locale = "fr_FR"

// call a method on the service
service.helloWorld()
```

This creates a SOAP header similar to the following:

```
<soap12:Envelope xmlns:soap12="http://www.w3.org/2003/05/soap-envelope">
<soap12:Header>
  <gwsoap:locale xmlns:gwsoap="http://guidewire.com/ws/soapheaders">fr_FR</gwsoap:locale>
</soap12:Header>
<soap12:Body>
  <getPaymentInstrumentsFor xmlns="http://example.com/gw/webservice/bc/bc700/PaymentAPI">
    <accountNumber>123</accountNumber>
  </getPaymentInstrumentsFor>
</soap12:Body>
</soap12:Envelope>
```

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

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 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.

IMPORTANT Be careful not to confuse one-way methods with asynchronous methods. For more information about asynchronous methods, see “Asynchronous Methods” on page 90.

Asynchronous Methods

Gosu supports optional asynchronous calls to web services. Gosu exposes alternate web service methods signatures on the service 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 (check regularly whether it is done) to choose to get results later (synchronously in relation to the calling code).

See the introductory comments in “Consuming Web Service Overview” on page 75 for related information about the basic types of connections for a method.

IMPORTANT Be careful not to confuse one-way methods with asynchronous methods. For more information about one-way methods, see “One-Way Methods” on page 89.

The `AsyncResponse` object contains the following properties and methods:

- `start` method – initiates the web service request but does not wait for the response
- `get` 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 is an example of calling the a synchronous version of a method contrasted to using the asynchronous variant of it.

```
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.

On a related topic, you can support MTOM support when publishing a web service. See “Web Service Invocation Context” on page 48.

General Web Services

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

Note: This topic relies on you to understand what web services are and how to call them from remote systems. Before reading this topic, read “What are Web Services?” on page 37.

This topic includes:

- “Mapping Typecodes to External System Codes” on page 93
- “Importing Administrative Data” on page 95
- “Maintenance Tools Web Service” on page 96
- “System Tools Web Services” on page 97
- “Workflow Web Services” on page 98
- “Profiling Web Services” on page 99

Mapping Typecodes to External System Codes

If possible, configure the PolicyCenter typelists to include typecode values that match those already used in external systems. If you can do that, you do not need to map codes between systems.

However, in many installations this is infeasible and you must map between internal codes and external system codes. For example, you might have multiple external legacy systems and they do not match each other, so PolicyCenter can only match one system at maximum.

PolicyCenter provides a built-in utility to support simple typecode mappings. The first step in using the utility is to define the mappings. The mappings go into the `PolicyCenter/modules/configuration/config/typelists/mapping/typecodemapping.xml` file. The following is a simple example:

```
<?xml version="1.0"?>
<typecodemapping>
  <namespacelist>
    <namespace name="ns1" />
    <namespace name="ns2" />
  </namespacelist>

  <typelist name="LossType">
```

```

<mapping typecode="PR" namespace="ns1" alias="Prop" />
<mapping typecode="PR" namespace="ns2" alias="CPL" />
</typelist>
</typecodemapping>

```

The first section of the mapping file lists the set of *namespaces*. These namespaces correspond to the different external systems you need to map. For example, if the mappings are different between two external systems, then each has its own namespace.

The rest of the mapping file contains sections for each typelist that requires mapping. Within the typelist, add elements for mapping entries. Each mapping entry contains:

- **Typecode code** – The PolicyCenter typecode to map.
- **Namespace** – The namespace is the name of the external system in the XML file and used by any tool that translates type codes. You may wish to define your namespace strings with your company name and a system name. For example, for the company name ABC for a check printing service, you might use "ABC:checkprint".
- **Typecode alias** – The *alias* is the value for this typecode in the external system.

There can be multiple mapping entries for the same typecode and namespace. This supports situations in which multiple external codes map to the same PolicyCenter code during data import.

Using Web Services to Translate Typecodes

After you define the mappings, you can translate between internal and external codes using `TypeListToolsAPI` web service methods:

- `getAliasByInternalCode` – Find the alias, if any, for a given typelist, code, and namespace. If there is no alias for that code and namespace, returns `null`. The `null` return value indicates that the internal and external codes are the same.
- `getTypeKeyByAlias` – Find the internal typecode, if any, for a given typelist, alias, and namespace. This returns a data object that contains the typecode's code, name, and description properties. If no mapping is found, there are two possible meanings. It might indicate that the external and internal codes are the same. However, it might indicate that the mapping is missing from your mapping code. Use the `getTypeListValues` method to verify that the external code is a valid internal code in this situation.
- `getAliasesByInternalCode` – During exports, gets an array of `String` values that represent external aliases to internal typecodes given a typelist, a namespace, and an internal code.
- `getTypeKeysByAlias` – During imports, gets an array of `TypeKeyData` objects given a typelist, a namespace, and an alias.
- `getTypeListValues` – Given the name of a typelist, returns an array of all the typekey instances contained within that typelist.

From Gosu and Java code that runs on the PolicyCenter server, always use the `TypecodeMapperUtil` utility class, not the `TypeListToolsAPI` web service. See “Using Gosu or Java to Translate Typecodes” on page 94.

Using Gosu or Java to Translate Typecodes

Typecode translation may occur very frequently in Gosu or Java plugin code, or in Gosu templates used for data extraction by external systems. From Gosu and Java code that runs on the PolicyCenter server, always use the `TypecodeMapperUtil` utility class, not the `TypeListToolsAPI` web service.

From Gosu or Java, translate a typecode using `TypecodeMapperUtil`. Call its static method `getTypecodeMapper` to return a mapper object that has a `getInternalCodeByAlias` method.

From Gosu, the code looks like:

```

var mapper = gw.api.util.TypecodeMapperUtil.getTypecodeMapper()

var mycode = mapper.getInternalCodeByAlias( "Contact", "ABC:system1", "ATTORNEY" )

```

See also

- For details of other plugin utilities, see “Useful Java Plugin APIs” on page 131.

Importing Administrative Data

PolicyCenter provides tools for importing and exporting data in XML using the import tools API (`ImportToolsAPI`) web service. The easiest way to export data suitable for import is to use the built-in user interface in the application.

For related topics, see:

- “Importing Administrative Data Using the `import_tools` Command” on page 104 in the *System Administration Guide*
- “Importing and Exporting Administrative Data from PolicyCenter” on page 104 in the *System Administration Guide*

To prepare 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 these files by calling the `regen-xsd` command using the `gwpc` command line tool. See “Importing and Exporting Administrative Data from PolicyCenter” on page 104 in the *System Administration Guide*.

The `ImportToolsAPI` web service method `importXmlData` imports administrative data from an XML file. Only use this for administrative database tables. Any other use is unsafe. This API does not perform data validation on imported data. Administrative tables include `User` and `Group` and their related entities.

IMPORTANT The `ImportToolsAPI` web service method `importXmlData` is only supported for administrative data due to the lack of validation for this type of import.

The main XML import routine is `importXmlData` and it takes a single `String` argument containing XML data:

```
importToolsAPI.importXmlData(myXMLData);
```

CSV Import and Conversion

There are several other methods in this interface related to importing and converting to and from comma-separated value data (CSV 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 these CSV-related 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 occur. Do not try to import or export all administrative data in a dataset at once.

Maintenance Tools Web Service

The maintenance tools (`MaintenanceToolsAPI`) web service provides a set of tools available only if the system is at the `maintenance run` level or higher.

Running Batch Processes Using Web Services

One of the methods in the `MaintenanceToolsAPI` web service is `startBatchProcess`, which runs a batch process or a writer for a work queue. The API notifies the caller that the request is received. The caller must poll the server later to see if the process failed or completed successfully. For server clusters, batch process runs only on the batch server. However, you can make the API request to any of the servers in the cluster. If the receiving server is not the batch server, the request automatically forwards to the batch server.

For example, run a batch process and get the process ID of the batch process:

```
processID = maintenanceTools.startBatchProcess("memorymonitor");
```

Terminate a batch process by process name or ID, for example:

```
maintenanceTools.terminateBatchProcessByName("memorymonitor");
maintenanceTools.terminateBatchProcessByID(processID);
```

Check the status of a batch process, for example:

```
maintenanceTools.batchProcessStatusByName("memorymonitor");
maintenanceTools.batchProcessStatusByID(processID);
```

For work queues, the status methods returns 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 may remain unprocessed.

For the batch process methods of the maintenance tools API, the batch processes apply only to PolicyCenter, not additional Guidewire applications that you integrated with it.

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. The typecode must include the `APIRunnable` category to start your custom batch process with the `MaintenanceToolsAPI` web service.

See also

- For the list of codes for batch processes in the base configuration, see “List of Work Queues and Batch Processes” on page 124 in the *System Administration Guide*
- For alternatives to the `MaintenanceToolsAPI` web service, see “Running Work Queue Writers and Batch Processes” on page 115 in the *System Administration Guides*.

Manipulating Work Queues Using Web Services

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 APIs query or modify the existing work queue configuration. For example, APIs can get the number of the workers threads configured for this server (`instances`) and the configured delay between processing each work item (`throttleInterval`).

Wake up all workers for the specified work queue across the entire cluster:

```
maintenanceTools.notifyQueueWorkers("ActivityEsc");
```

Get the work queue names for this instance of PolicyCenter:

```
stringArray = maintenanceTools.getWorkQueueNames();
```

Get the number of instances and throttle interval for a work queue:

```
WorkQueueConfig wqConfig = maintenanceTools.getWorkQueueConfig("ActivityEsc");
numInstances = wqConfig.getInstances();
```

Set the number of instances and throttle interval for a work queue:

```
WorkQueueConfig wqConfig = new WorkQueueConfig;  
wqConfig.setInstances(1);  
wqConfig.setThrottleInterval(999);  
WorkQueueConfig wqConfig = 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 using the batch process web service API 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 term *product* and *cluster* apply to the current Guidewire product only as determined by the SOAP API server requested.

Stopping Startable Plugins Using Web Services

A *startable plugin* is a special type of code that runs without human intervention in the application server as a background process, beginning at server startup. You can stop startable plugins and start them again by using methods on the `MaintenanceToolsAPI` web service.

To stop a startable plugin, call the `stopPlugin` method. To start a startable plugin again, call the `startPlugin` method. These methods take the plugin name in `String` format as their only argument. Plugin names are defined in the Plugin Registry in Studio.

In general, startable plugins run only the batch server, but you can develop startable plugins that run distributed on all servers. Before you use the `MaintenanceToolsAPI` web service to start or stop a distributed startable plugin, be certain you understand the implications for state management across all servers.

See also

- “Startable Plugins” on page 259
- “Using the Plugins Registry Editor” on page 131 in the *Configuration Guide*
- “Configuring Startable Plugins to Run on All Servers” on page 263

System Tools Web Services

The system tools API (`SystemToolsAPI`) interface provides a set of tools that are always available, even if the server is set to `dbmaintenance` 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 Studio.

Getting and Setting the Run Level

The most important usage of the `SystemToolsAPI` interface is to set the system run level:

```
systemTools.setRunLevel(SystemRunlevel.GW_MAINTENANCE);
```

Or, to get the system run level:

```
runlevelString = systemTools.getRunLevel().getValue();
```

Sometimes you may want a more lightweight way of determining the run level of the server from another computer on the network than to use SOAP APIs. You might want to informally use your web browser during development to check the run level.

To check the run level, simply call the ping URL on a PolicyCenter server:

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

For example:

```
http://PolicyCenter.mycompany.com:8080/pc/ping
```

Assuming that the server is running, this returns an extremely short result as an HTML document containing a text encoding of the server run level in a special format.

If you are just checking whether the server is up, you do not care about the return result from this ping URL. Typically in such cases, if it returns a result at all it proves your server is running. In contrast, if there is an HTTP error or browser error, the server is not running to respond to the ping request.

If you want the actual run level, check the contents of the HTTP result. However, the value is not simply a standard text encoding of the public run level enumeration. It represents the ASCII character with decimal value of an integer that represents the internal system run level (30, 40, 50).

The following table lists the correlation between the run level and the value return by the ping URL:

Run level	Ping URL result character
<i>Server not running</i>	<i>No response to the HTTP request</i>
DB_MAINTENANCE	ASCII character 30, which is the Record Separator character
MAINTENANCE	ASCII character 40, which is the character "("
MULTIUSER	ASCII character 50, which is the character "2"
GW_STARTING	ASCII character 0. A null character result might not be returnable for some combinations of HTTP servers and clients.

Getting Server and Schema Versions

You can use the `SystemToolsAPI` interface to get the current server and schema versions.

The following example code in Java demonstrates how to get this information:

```
versionInfo = systemTools.getVersion();
appVersion = versionInfo.getAppVersion();
schemaVersion = versionInfo.getSchemaVersion();
configVersion = versionInfo.getConfigVersion();
configVersionModified = versionInfo.getConfigVersionModified();
```

Workflow Web Services

The web service API interface `WorkflowAPI` allows you to control PolicyCenter workflows from external client API code, including PolicyCenter plugins that use the web service APIs. In addition to being called by remote systems, the built-in `workflow_tools` command-line tools use these methods internally.

Workflow Basics

You can invoke a workflow trigger remotely from an external system using the `invokeTrigger` method. To check whether you can invoke that trigger, call the `isTriggerAvailable` method, described later in the section.

Be aware that any time the application detects a workflow error, the workflow sets itself to the state `TC_ERROR`. If this happens, you can remotely resume the workflow using these APIs.

Refer to the following table for workflow actions you can request from remote systems:

Action	WorkflowAPI method	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.

Profiling Web Services

From remote systems you can enable or disable the PolicyCenter profiler system using the ProfilerAPI web service. The methods use these method arguments:

- A boolean value that indicates whether to enable (`true`) or disable (`false`) the profiler for a particular component of the system, which varies by method.
- A process type (a typecode in the BatchProcessType typelist)
- A boolean value that controls whether to use high resolution clock for timing (`true`) or not (`false`). This only has an affect on the Windows operating system.
- A boolean value that controls whether to enable stack traces (`true`) or not (`false`). This feature is performance intensive, so think carefully before enabling this. This parameter is ignored if the first argument (to enable profiling) is false.
- A boolean value that controls whether to enable query optimizer tracing (`true`) or not (`false`). This feature is performance intensive, so think carefully before enabling this. This parameter is ignored if the first argument (to enable profiling) is false.
- A boolean value that controls whether to allow *extended* query tracing. This feature is performance intensive, so think carefully before enabling this. This parameter is ignored if the first argument (to enable profiling) is false.
- An integer (`int`) value that threshold for how long (in milliseconds) a database operation can take before generating a report using dbms counters. Set the value 0 to disable dbms counters.

Batch Processes

To enable the profiler for batch processes of a specific type, call the `setEnableProfilerForBatchProcess` method.

Batch Processes and Work Queues

To enable the profiler for batch processes and work queues, call the `setEnableProfilerForBatchProcessAndWorkQueue` method.

Messaging Destinations

To enable the profiler for messaging destinations, call the `setEnableProfilerForMessageDestination` method.

Startable Plugins

To enable the profiler for startable plugins, call the `setEnableProfilerForStartablePlugin` method.

Subsequent Web Sessions

To enable the profiler for subsequent web sessions, call the `setEnableProfilerForSubsequentWebSessions` method.

Web Services

To enable the profiler for web services, call the `setEnableProfilerForWebService` method.

Work Queues

To enable the profiler for work queues, call the `setEnableProfilerForWorkQueue` method.

Account and Policy Web Services

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

This topic includes:

- “Account Web Services” on page 102
- “Job Web Services” on page 104
- “Policy Cancellation and Reinstatement Web Services” on page 105
- “Submission Web Services” on page 107
- “Producer Web Services” on page 108
- “Product Model Web Services” on page 109
- “Policy Web Services” on page 111
- “Policy Period Web Services” on page 112
- “Policy Earned Premium Web Services” on page 113
- “Import Policy Web Services” on page 113
- “Policy Change Web Services” on page 114
- “Policy Renewal Web Services” on page 115
- “Archiving Web Services” on page 119
- “Quote Purging Web Services” on page 119

See also

- For general information about web services and how to call them from external systems, see “Web Services Introduction” on page 37.
- For information about web services related to claim system integration, including integration with ClaimCenter, see “Claim and Policy Integration” on page 507.

Account Web Services

Use the AccountAPI web service to add documents to accounts, add notes to accounts, and find account public IDs. This web service is WS-I compliant.

Adding 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.

Adding 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.

Finding 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. For example:

```
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];
```

Assigning 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 an account public ID `String`, an activity type (`ActivityType`), an activity pattern code (`String`), an activity pattern code (`String`), and an array of activity properties (`ActivityField[]`). This method returns nothing.

Adding Contacts to an Account

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

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

Adding Location to an Account

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

Getting Account Numbers

You can remotely get the location to an account by using the `getAccountNumber` method. It takes an account public ID (`String`) and returns the account number (`String`).

Checking 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 (real-world) date and time.

The method takes an account public ID (`String`) and returns `true` if the account has active policies, and `false` otherwise.

Inserting Accounts

You can remotely insert an account by using the `insertAccount` method. It takes 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 public ID (`String`) of the new `Account` entity.

Merging 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 account public IDs (`String` values). 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.

You can customize the behavior after merging accounts, for example to specially handle data model extensions during the merge. For more details, see “Account Plugin Details” on page 151.

Transferring Policies

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

You can customize the behavior after transferring accounts, for example to specially handle data model extensions during the merge. For more details, see “Account Plugin Details” on page 151.

Updating Account Contacts

You can remotely update the contact information for account contacts by using the `updateAccountContact` method. You provide an account public ID (`String`), a contact display name (`String`), and account contact information encapsulated in an `AccountContactInfo` entity. This 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. This web service is WS-I compliant.

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.

See also

- “Activity Patterns” on page 372 in the *Application Guide*.

Common Parameters in the Job APIs

The common parameters for the primary methods are:

- Public ID of the job, as a `String`.
To get the public ID, use the helper method `findJobPublicIdByJobNumber`.
- Public ID of the activity pattern, as a `String`.
To get the public ID, use the method
`ActivityPattern.finder.getActivityPatternByCode("pattern-code")`.
- Activity fields, as a `gw.webservice.pc.pc700.gxmodel.activitymodel.types.complex.Activity`.
Use an instance of the XML type `Activity` to initialize activity fields that activity patterns do not cover. For more information, see “The Guidewire XML (GX) Modeler” on page 306 in the *Gosu Reference Guide*.

Note the following about 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. For more information, see “Preempted Jobs” on page 513 in the *Application Guide*.

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

For more information about circumstances in which PolicyCenter might trigger cancellation or reinstatement, refer to “Policy Period Management” on page 492.

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.

Beginning a Cancellation

An external system can begin a cancellation in PolicyCenter with the `beginCancellation` method on CancellationAPI. This web service is WS-I compliant.

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

This method returns the public ID of the cancellation job.

The method signature looks like:

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

The parameters are:

- Policy number, as a `String`
- Cancellation date, as a `Calendar` object
- Recalculate effective date (`recalculateEffDate`), as a boolean. See discussion after this list.

- Cancellation source typecode, `carrier` (carrier initiated cancel) or `insured` (the insured initiated cancel)
- Reason code typecode, as a `ReasonCode` value. Refer to plugin API doc or Data Dictionary for details.
- 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")
```

Rescinding a Cancellation

Rescind a cancellation with the `rescindCancellation` method. This web service is WS-I compliant.

This 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 carrier initiated the cancellation with the value `carrier` or the insured initiated the cancellation with the value `insured`.

The method signature for this method is:

```
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")
```

Finding 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. This web service is WS-I compliant. 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 carrier
 - `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);
```

Reinstating a Policy

Use the `ReinstateAPI` 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 one method, `beginReinstatement`, which has the following arguments:

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

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

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

```
reinstateAPI.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. This web service is WS-I compliant.

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 using the Guidewire XML model (GX model) definition. The GX Model is available at the location `gw.webservice.pc.pc800.gxmodel.policyperiodmodel`. Then the code calls `populatePolicyPeriod` enhancement method on the XML object. The enhancement method is implemented in `gw.webservice.pc.pc800.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.

The method signature is:

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

The parameters are:

- 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.

The method signature is:

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

The parameters are:

- 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. This web service is WS-I compliant.

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

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

Remember the following 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 (non-production) 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 manage the product model by using Product Designer.
- Update system tables on a running development (non-production) server based on local file system representation of the system tables.
- Query the server for product model information.

This web service is WS-I compliant.

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 *workspaces* (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 fundamental difference between working with a development server (no genuine customer data) and a production server (customer data bound by legal and customer service implications).

The development server must have an identical PolicyCenter configuration as the production server except for the following:

- 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`.

WARNING Fully test product model changes on a development (non-production) 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 lookup type name, as a `String`
 - `PolicyLinePatternCode` – the policy line pattern code, as a `String`
 - `CovTermPatternCode` – the coverage term pattern code, as a `String`
 - `ProductCode` – the product code, as a `String`
 - `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` – an industry code, as a `String`
 - `VehicleType` – a vehicle type, as a `VehicleType` object
- `offeringCode` – An offering code, as a `String`
- `lookupDate` – the date to look up, as a standard `Date` object

Exceptions that the methods may throw:

- `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. This web service is WS-I compliant.

Adding 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.

The method signature is:

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

The parameters include:

- 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.

Closing a Referral Reason

There is also a method called `closeReferralReason` which closes a referral reason from an external system.

The method signature is:

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

The parameters include:

- a policy period public ID
- a code that matches the `UWIssueType.Code` of the referral reason to close
- an identifier for the referral reason (an issue key)

The method returns nothing.

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`.

The syntax of this API is as follows:

```
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 modeler (GX). See “The Guidewire XML (GX) Modeler” on page 306 in the *Gosu Reference Guide*.

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 has the following behavior:

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 `EscalationInc1Days`. 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, you can use a variant of the `addActivityFromPatternAndAutoAssign` method. Instead, call the `addActivityFromPatternAndAssignToUser` method. It takes the same arguments but with additional arguments for a user ID and a group ID:

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

The behavior is the same as the steps listed in “Add Activity and Auto-Assign” on page 111, except that it assigns the activity 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, you can use a variant of the `addActivityFromPatternAndAutoAssign` method. Instead, call the `addActivityFromPatternAndAssignToUser` method. It takes the same arguments but with an additional argument for a queue ID (`queueId`):

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

The behavior is the same as the steps listed in “Add Activity and Auto-Assign” on page 111, except that it assigns the activity a queue instead of auto-assignment.

Policy Period Web Services

To add notes to policies and policy periods and to add documents to policy periods, use the `PolicyPeriodAPI` web service. This web service is WS-I compliant.

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 Guidewire XML models for notes or documents, as `String` parameters. For more information, see “The Guidewire XML (GX) Modeler” on page 306 in the *Gosu Reference Guide*.

Adding 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 Guidewire XML model of a note, as a `gw.webservice.pc.pc700.gxmodel.notemodel.types.complex.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 Guidewire XML model of a note, as a `gw.webservice.pc.pc700.gxmodel.notemodel.types.complex.Note`.

Adding Documents to Policy Periods

Use the `addDocumentToPolicyPeriod` method to add a document to a policy period. It takes the public ID of a policy period and a Guidewire XML model of a document, as a `gw.webservice.pc.pc700.gxmodel.documentmodel.types.complex.Document`.

Policy Earned Premium Web Services

To calculate earned premiums from an external system, use the web service `PolicyEarnedPremiumAPI`. This web service is WS-I compliant. It has one method, `calcEarnedPremiumByPolicyNumber`, which calculates the earned premium for a given policy number. The method takes the following arguments:

- a policy number, as a `String`
- a date that represents the period, as a `Date` object
- a date as of which to calculate earned premium amounts
- a boolean value that specifies whether to include data that is *earned but not reported* (EBUR).

Regarding 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 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

Import Policy Web Services

To import policies from XML data from an external system, use the `PolicyChangeAPI` web service. This web service is WS-I compliant.

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

The `quoteSubmission` method takes 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 be in the format defined by the Guidewire XML modeler (GX model) in the file:

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

The method returns an object of type `QuoteResponse`, which is a Gosu class that contains properties for:

- `JobNumber` – the 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 . This web service is WS-I compliant.

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. For more information, see “Policy Change Transaction” on page 115 in the *Application Guide*.

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

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

To handle requests about policy renewals from external systems, use the policy renewal web service `PolicyRenewalAPI`. The `PolicyRenewalAPI` web service is in the package `gw.webservice.pc.VERSION.job`. This web service is WS-I compliant. The policy renewal web service provides methods for:

- “Starting Renewals on Existing Policies” on page 115
- “Importing a Policy for Renewal” on page 115
- “Policy Renewal Methods for Billing Systems” on page 118

Starting Renewals on Existing Policies

The policy renewal web service provides the `startRenewals` method to start renewals on policies that already exist in PolicyCenter. The policy numbers are specified in an array of `String` objects. The method handles the policy numbers in sequence. Each successful renewal commits to the database as it is created. There is no gap in coverage between the renewal and its based-on policy period.

The `startRenewals` method has a `policyNumbers` parameter. This parameter is an array of policy numbers as `String` objects. Each policy number must match a policy in the PolicyCenter database.

This method throws an exception if a renewal does not succeed. The remaining policies are not processed.

Importing a Policy for Renewal

The policy renewal web services provides the `startConversionRenewal` method which imports a policy renewal from a hypothetical external system. The method creates a new policy in an existing account, and creates a policy period on the policy. The method starts a renewal based on this policy period. There is no gap in coverage between the renewal and its based-on policy period.

In the default configuration, the renewal data from the external system is in XML format. You can modify the `startConversionRenewal` method to handle your input data. You can customize the XML format to match your implementation of policy periods and subobjects.

Note: The included web service Gosu code is intended as an example and a starting point for your own conversion code. You must modify the example conversion code for your business use.

Considerations when Importing Policy Renewals

Be aware of the following considerations when importing policy renewals using the `startConversionRenewal` method:

- The policy must not already exist in PolicyCenter.
- In each request, you can pass only one policy period in the `policyPeriodData` argument. Do not attempt to modify the example implementation of the `startConversionRenewal` method to support multiple policy renewals in a single request.
- The account for the policy must exist already in PolicyCenter. Any account creation or updates must be done separately, before calling this method. For methods to insert and update accounts from an external system, see “Account Web Services” on page 102.
- Reference data must already exist in PolicyCenter. Reference data includes entities such as `ProducerCode` and `UWCompany`. Reference these entities by their public ID `String` values.
- The `startConversionRenewal` method does not support `Note` or `Document` objects. Normally, these objects do not directly attach to a `PolicyPeriod` or `Policy`. Instead, notes and documents have foreign keys that point back to their owners. Another web service provides methods exist for adding notes and documents to `PolicyPeriod` objects. See “Policy Period Web Services” on page 112.

Create a Policy Period and Start the Renewal

The `startConversionRenewal` method creates a new policy on an existing account, and then creates a policy period associated with that policy. The method creates and starts a renewal based on the policy period.

The `startConversionRenewal` method takes the following parameters:

- `accountNumber` – The account number, as a `String`.
- `productCode` – The code of the product, as a `String`. For example, `PersonalAuto` or `WorkersComp`.
- `producerCodeId` – The public ID, as a `String`, of the producer code.
- `policyNumber` – The policy number, as a `String`, for the policy periods associated with this renewal. If `null`, generate a new, unique `policyNumber`. Raise an underwriter issue if not unique.
- `policyPeriodData` – The data, as a `String`, to populate the new policy period.
- `changesToApply` – Changes, as a `String`, to make in the renewal period that differ from the legacy period. This parameter is not used in the default configuration.
- `parseOptions` – The options, as a `String`, passed to the XML parser to parse the `policyPeriodData`.
- `basedOnEffectiveDate` – The effective date, as a `String`, for the based-on policy period.
- `basedOnExpirationDate` – The expiration date, as a `String`, for the based-on policy period. This is also is the effective date of the new renewal period.

This method returns the job number of the newly started renewal.

The `policyPeriodData` parameter is a `String` representation of a policy renewal from the external system. This parameter includes data for creating the policy, policy period, and renewal. In the default implementation, the `policyPeriodData` must be in XML format as defined by the `PolicyPeriodModel` Guidewire XML (GX) model format. In Studio, the model is in the `gw.webservice.pc.pc700.gxmodel` package.

In the default implementation, the `startConversionRenewal` method parses the `policyPeriodData` parameter as XML using a Guidewire XML model (GX model) definition. The GX Model is available at the location `gw.webservice.pc.pc800.gxmodel.policyperiodmodel`. Then the code calls `populatePolicyPeriod` enhancement method on the XML object. The enhancement method is implemented in `gw.webservice.pc.pc800.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.

The `startConversionRenewal` method calls the `account.createConversionRenewalWithBasedOn` method to create the policy and based-on policy period, and to start the renewal. The `createConversionRenewalWithBasedOn` method commits the policy and based-on policy period to the database. In a separate bundle, the `createConversionRenewalWithBasedOn` method commits the renewal to the database. The renewal is in draft mode and is not bound.

The `startConversionRenewal` method returns the renewal job number in a `String`.

You can customize the `startConversionRenewal` method to handle additional processing logic for policy renewals initiated from your external system.

See also

- “The Guidewire XML (GX) Modeler” on page 306 in the *Gosu Reference Guide*

About the New Policy

The `startConversionRenewal` method creates a new policy and commits the new policy to the database.

The policy graph includes the following information:

- A new `Policy` that links to the target `Account` with the policy number specified. Creates an underwriting issue if the policy number is already in use.

- The policy's `Product` property set to the `productCode` parameter of the method.
- The policy's loss history type (`LossHistoryType`) field defaults to the value that represents no loss history (NOL).

To create the policy, the `startConversionRenewal` method creates by calling the `account.createConversionRenewalWithBasedOn` method.

About the Based-on Policy Period

This topic describes the based-on policy period that the `startConversionRenewal` method creates and commits to the database.

In the new policy, PolicyCenter creates exactly one new based-on policy period entity instance:

- The `createConversionRenewalWithBasedOn` method sets the policy period's primary named insured and primary location from the target account's account holder and primary location respectively.
- The method sets the policy period's period start date (`PeriodStart`) and the edit effective date (`EditEffectiveDate`) to the `basedOnEffectiveDate` parameter for this method. If the `basedOnEffectiveDate` parameter is `null`, then the method throws an exception.
- The method sets the period end date (`PeriodEnd` property) of the policy period to the `basedOnPeriodExpirationDate` parameter. If the `basedOnPeriodExpirationDate` is `null`, then the method throws an exception.
- The `basedOnEffectiveDate` or `basedOnPeriodExpirationDate` are passed as `String` objects. If either value cannot be parsed as a valid date, then the method throws an exception.
- The `createConversionRenewalWithBasedOn` method uses the effective time plugin (`IEffectiveTimePlugin`) to set the time component of the policy period's `PeriodStart` and `PeriodEnd` properties. The `createConversionRenewalWithBasedOn` method gets the time from the `IEffectiveTimePlugin` plugin by using the `getConversionRenewalEffectiveTime` and `getConversionRenewalExpirationTime` methods. Then the `createConversionRenewalWithBasedOn` method sets the time to these values. The date component of the `datetime` value from effective time plugin is ignored. In the default configuration, the `EffectiveTimePlugin` plugin implements this interface. In the `EffectiveTimePlugin`, the `createConversionRenewalWithBasedOn` method expects the caller to provide exact `datetime` values or uses the `DEFAULT_TIME_STRING` time value from the effective time plugin. For more information, see "Effective Time Plugin" on page 161.
- The based-on policy period's `TermType` is `Other`.
- The based-on policy period has a `PolicyPeriodStatus` set to `LegacyConversion`.
- The method creates appropriate `PolicyLine` objects on the `PolicyPeriod`, based on the `Product` definition. Each `PolicyLine` has its `BaseState` property set to the state of the primary named insured's primary address. The `initialize` method in the matching `PolicyLineMethods` Gosu class runs. You can modify the base state with additional policy line logic. Depending on the line of business, the `initialize` method can sync modifiers, create coverages, to initialize line-specific auto-number sequences, or other actions.
- The method sets the `LocationAutoNumberSeq` property on the policy period to a new `AutoNumberSequence` entity instance. Because of this, location auto-numbering (including the primary location mentioned above) begins at number one.
- The method creates new and empty `PolicyTerm` and `EffectiveDatedFields` entity instances that link to the policy period.

The `startConversionRenewal` method creates the based-on policy period by calling the `account.createConversionRenewalWithBasedOn` method. The `createConversionRenewalWithBasedOn` method commits the policy period to the database.

About the New Renewal

The `startConversionRenewal` method creates a renewal job and commits the job to the database. The renewal is in draft mode and is not bound.

The renewal has the following information:

1. The renewal's `CloseDate` property is `null`.
2. The new renewal job has exactly one `UserRoleAssignment` object. On the `UserRoleAssignment` object, the `AssignedUser` property is the current user. The `Role` property has the value `Creator`.
3. The `createConversionRenewalWithBasedOn` method generates a job number for the renewal.
4. The `createConversionRenewalWithBasedOn` method calls the `start` method in the renewal process (`RenewalProcess`).

The `startConversionRenewal` method creates the renewal job by calling the `account.createConversionRenewalWithBasedOn` method.

Policy Renewal Methods for Billing Systems

Billing systems use the `PolicyRenewalAPI` web service to send confirmation of renewals to PolicyCenter. This web service is WS-I compliant. The web services is implemented in Gosu as `gw.webservice.pc.pc700.job.PolicyRenewalAPI.gs`, for integration with BillingCenter 7.0, or `gw.webservice.pc.pc400.job.PolicyRenewalAPI.gs`, for integration with BillingCenter 4.0.

You can configure PolicyCenter renewals to operate with or without renewal confirmation. For more information, see “Billing Implications of Renewals or Rewrites” on page 476.

Confirm a Policy Term

If you configure PolicyCenter to operate with renewal confirmations, your billing system can notify PolicyCenter that an insured has confirmed a renewal. If a customer submits a payment for a policy renewal, the payment confirms the renewal. In the `PolicyRenewalAPI` web service, the `confirmTerm` method confirms a policy term by its policy number and term number. A billing system calls this method whenever the billing system receives a renewal payment.

The `confirmTerm` method takes the following parameters:

- `policyNumber` – The policy number that contains the term, as a `String`.
- `termNumber` – The term number to confirm, as an `int`.
- `transactionId` – The unique identifier for this request, as a `String`.

This method updates the policy period confirmed status in the billing system by calling the `updatePolicyPeriodConfirmed` method on the billing system plugin defined by the `IBillingSystemPlugin` plugin interface.

This method throws an exception if:

- It does not find a matching policy and term number.
- The policy term is archived.
- The policy term is not bound.

Notify PolicyCenter of Receipt of Payment for Renewal

The policy renewal web service provides the `notifyPaymentReceivedForRenewalOffer` method to notify PolicyCenter of receipt of payment for a specific renewal offer. The method takes the following parameters:

- `offerID` – The unique identifier of the renewal offer, as a `String`.
- `selectedPaymentPlanCode` – The selected payment plan code, as a `String`.

- `paymentAmount` – The payment amount, as a `MonetaryAmount`.
- `transactionId` – The unique identifier for this request, as a `String`.

This `notifyPaymentReceivedForRenewalOffer` creates an activity if:

- The policy is not in the correct state to receive this payment.
- No applicable payment plan for this payment amount exists.

If PolicyCenter can apply the payment, the `notifyPaymentReceivedForRenewalOffer` method returns the new policy number as a `String`. If PolicyCenter cannot apply the payment, the method returns `null`.

Archiving Web Services

To manipulate archiving from external systems, call the `ArchiveAPI` web service. This web service is WS-I compliant. For more information about archiving integration, see “Archiving Integration” on page 557.

Check If a Policy Term is Archived

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

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 takes a `String` It takes a policy number as a `String` and an effective date. This API first determines which policy term is effective at the given date. Next, it marks that policy term as a term to restore.

Suspending and Resuming Archiving

PolicyCenter includes a feature to mark a policy as ineligible for additional archiving. This feature is sometimes known as *suspending archiving*. There are three methods in `ArchiveAPI` that relate to suspending archiving. All three methods take only a single argument, which is the policy number of the policy in question. The methods let you:

- **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. This web service provides methods for flagging whether to purge a policy or policy period. This web service is WS-I compliant.

See also

- “Quote Purging” on page 449 in the *Application Guide*
- “Configuring Quote Purging” on page 477 in the *Configuration Guide*

Do Not Purge Flag on a Policy Period

The PurgeAPI web service has methods related to the `DoNotPurge` flag on a policy period. These methods take as input the `PublicID` of the policy period. The methods let you:

- **Check whether a policy period is excluded from purge** – From an external system, call the `isDoNotPurgePolicyPeriod` method to check the `DoNotPurge` flag on a policy period.
- **Flag a policy period as excluded from purge** – From an external system, call the `setDoNotPurgePolicyPeriod` method to set the `DoNotPurgeFlag` on a policy period to `true`.
- **Flag a policy period as not excluded from purge** – From an external system, call the `unsetDoNotPurgePolicyPeriod` method to set the `DoNotPurgeFlag` on a policy period to `false`.

part III

Plugins

Plugin Overview

PolicyCenter plugins are software modules that PolicyCenter calls to perform an action or calculate a result. PolicyCenter defines a set of plugin interfaces. You can write your own implementations of plugins in Gosu or Java.

This topic includes:

- “Overview of PolicyCenter Plugins” on page 124
- “Error Handling in Plugins” on page 129
- “Temporarily Disabling a Plugin” on page 129
- “Example Gosu Plugin” on page 129
- “Special Notes For Java Plugins” on page 130
- “Getting Plugin Parameters from the Plugins Registry Editor” on page 131
- “Writing Plugin Templates For Plugins That Take Template Data” on page 132
- “Plugin Registry APIs” on page 134
- “Plugin Thread Safety” on page 135
- “Reading System Properties in Plugins” on page 139
- “Do Not Call Local Web Services From Plugins” on page 140
- “Creating Unique Numbers in a Sequence” on page 140
- “Restarting and Testing Tips for Plugin Developers” on page 141
- “Summary of All PolicyCenter Plugins” on page 141

See also

- To help with writing Java plugins, see “Calling Java from Gosu” on page 123 in the *Gosu Reference Guide*.
- To help understand plugin interfaces, see “Interfaces” on page 215 in the *Gosu Reference Guide*.
- For information about messaging plugins, see “Messaging and Events” on page 289.
- For information about authentication plugins, see “Authentication Integration” on page 181.
- For information about document and form plugins, see “Document Management” on page 191.
- For information about other plugins, see “Other Plugin Interfaces” on page 251.

Overview of PolicyCenter 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:

- 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 easiest 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. For extended documentation on Java and OSGi usage, see “Java and OSGi Support” on page 629.

From a technical perspective, PolicyCenter defines a plugin as an *interface*, which is a set of functions (also known as 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. For conceptual information about interfaces, see “Interfaces” on page 215 in the *Gosu Reference Guide*.

Conceptually, there are two main steps to implement a plugin:

1. **Write a class (in Gosu or Java) that implements a plugin interface** – See “Implementing Plugin Interfaces” on page 124.
2. **Register your plugin implementation class** – See “Registering a Plugin Implementation Class” on page 126

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. For the maximum supported implementations for each interface, see the table in “Summary of All PolicyCenter Plugins” on page 141.

The plugin itself might do most of the work or it might consult with other external systems. Many plugins typically 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:

- **Gosu plugin** – A Gosu class. Because you write and debug your code directly in PolicyCenter Studio, in many cases it is easiest to implement a plugin interface as a Gosu class. The Gosu language has powerful features including type inference, easy access to web service and XML, and language enhancements such as Gosu blocks and collection enhancements.

- **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 after changes to data model configuration. You can use any Java IDE. 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.
- **OSGi plugin** – 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. To simplify OSGi configuration, PolicyCenter includes an application called IntelliJ IDEA with OSGi Editor. For more information about OSGi, see “Overview of Java and OSGi Support” on page 629.

IMPORTANT Guidewire recommends OSGi for all new Java plugin development.

The following table compares the types of plugin implementations you can create:

Features of each plugin implementation type	Gosu plugin	Java plugin (no OSGi)	OSGi plugin (Java with OSGi)
Choice of development environment			
You can use PolicyCenter 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 Your Plugin Implementation Class

First you must decide which plugin implementation type you want to use and launch the appropriate IDE based on whether you are using Gosu or Java. To learn how the plugin type affects what IDE you must use, see “Implementing Plugin Interfaces” on page 124.

In your IDE, create the class within your own package hierarchy such as `mycompany.plugins`. Do not create your own classes in the `gw.*` or `com.guidewire.*` package hierarchies.

In Gosu and Java, create a class with a declaration containing the `implements` keyword, such as:

```
public class MyContactSystemClass implements IContactSystemPlugin {
```

If your class does not properly implement the plugin interface, your IDE shows compilation errors and can offer to add methods to your class that the interface requires. You must fix any issues before you code compiles.

Implement all public methods of the interface. Create as many other private methods and related classes as you need to provide internal logic for your code. However, PolicyCenter only calls the public methods in your main implementation class.

IMPORTANT If you use OSGi, you must perform additional configuration. See “OSGi Plugin Deployment with IntelliJ IDEA with OSGi Editor” on page 647.

For Gosu Plugins, the Method Signatures May Be Different than Java

In PolicyCenter Studio or in your Java IDE, the compiler can detect that a class that implements an interface does not yet implement all the methods in the interface. The editor provides a tool that creates new stub versions of any unimplemented methods.

Note that the Gosu versions of the methods look different from the Java versions in many cases. The most common compilation issue is that if an interface's method looks like properties, you must implement the interface as a Gosu property.

If the interface contains a method starting with the substring `get` or `is` and takes no parameters, define the method using property syntax. Do not simply implement it as simple method with the name as defined in the interface. For example, if plugin `ExamplePlugin` declared a method `getMyVar()`, your Gosu plugin implementation of this interface must not include a `getMyVar` method. Instead, it must look similar to the following:

```
package mycompany.plugins

class MyClass implements gw.plugin.IExamplePlugin {

    property get MyVar() : String {
        ...
    }
}
```

Similarly, methods that begin with `set` and take exactly one argument become property setters.

See “Java get/set/is Methods Convert to Gosu Properties” on page 125 in the *Gosu Reference Guide*.

Plugin Templates (Only for Some Plugin Interfaces)

A small number of plugin interface methods provide method arguments that specify data as `String` values that contain data extracted from potentially large object graphs. This `String` data is called *template data*. Template data is the output of a Gosu template called a *plugin template*. A Gosu template is a short Gosu program that generates some output. In this case, the extracted data is an intermediary data format between rather than passing references to the original entity data objects directly to the plugin implementation.

If you see a plugin method with an argument called `templateData`, configure an appropriate plugin template that generates the information that your plugin implementation needs. See “Writing Plugin Templates For Plugins That Take Template Data” on page 132.

Built-in Plugin Implementation Classes

For some plugin interfaces, PolicyCenter provides a built-in plugin implementation that you can use instead of writing your own version. Some plugin implementation classes are pre-registered in the default 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.

PolicyCenter includes plugin implementations to connect with other Guidewire applications in the Guidewire InsuranceSuite. The plugin implementations for InsuranceSuite integration are located in packages that include the intended target application and version number. This helps ensure smooth migration and backwards compatibility among Guidewire applications. Carefully confirm package names for any plugin implementations you want to use. The package may include the Guidewire application two-digit abbreviation, followed by the application version number with no periods. For example, the shortened version of PolicyCenter 8.0.1 is `pc801`. Be sure to choose the plugin implementation class to match the version of the *other* external application, not the current application.

Registering a Plugin Implementation Class

In most cases, the first step to implementing a plugin interface is to create your implementation class. See “Implementing Plugin Interfaces” on page 124.

In other cases, you might be using a built-in plugin implementation class. See “Built-in Plugin Implementation Classes” on page 126.

In either case, you must also *register* the plugin implementation class so the application 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 use the Plugins registry, you must know all of the following:

- The plugin interface name. You must choose a supported PolicyCenter plugin interface as defined in “Summary of All PolicyCenter Plugins” on page 141. You cannot create your own plugin interfaces.
- The fully-qualified class name of your concrete plugin implementation class.
- Any initialization parameters, also called *plugin parameters*. See “Plugin Parameters” on page 127.

In nearly all cases, you must only have one active enabled implementation for each plugin interface. However, there are exceptions, such as messaging plugins, encryption plugins, and startable plugins. The table of all plugin interfaces has a column that specifies whether the plugin interface supports one or many implementations. See “Summary of All PolicyCenter Plugins” on page 141. If the plugin interface only supports 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*. 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. For example, enter the name `IContactSystemPlugin`.

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. See “Messaging Editor” on page 153 in the *Configuration Guide*. For encryption plugins, if you ever change your encryption algorithm, the plugin name is the unique identifier for each encryption algorithm.

To register a plugin, in Studio in the Project window, navigate to `configuration → config → Plugins → registry`. Right-click on `registry`, and choose `New → Plugin`. For more instructions about options in the Plugins Registry editor, see “Using the Plugins Registry Editor” on page 131 in the *Configuration Guide*.

In the dialog box that appears, it will ask for the interface. If the plugin name you used is the interface name (for example, `IContactSystemPlugin`), it is best to enter the interface name to be explicit. However, you can optionally leave the interface field blank.

IMPORTANT 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.

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. See “Getting Plugin Parameters from the Plugins Registry Editor” on page 131.

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. See “Using the Plugins Registry Editor” on page 131 in the *Configuration Guide*. For example, you could set a server name differently depending on whether you are running a development or production configuration.

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. For details, see “Using the Plugins Registry Editor” on page 131 in the *Configuration Guide*. If you do not specify a plugin directory, the default is `shared`.

IMPORTANT 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. To deploy OSGi bundles and third-party libraries in OSGi plugins, see “Java and OSGi Support” on page 629

For details of where to deploy your Java files for non-OSGi use, see “Deploying Non-OSGi Java Classes and JARs” on page 646.

Deploying Java Files (Including Java Code Called From Gosu Plugins)

How to deploy any Java files or third-party Java libraries varies based on your plugin type:

- 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. See “Deploying Non-OSGi Java Classes and JARs” on page 646. 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. See “For Java Plugins (Without OSGi), Define a Plugin Directory” on page 128. For details of where to put your Java files, see “Deploying Non-OSGi Java Classes and JARs” on page 646.
- For OSGi plugins (Java classes deployed as OSGi bundles), deployment of your plugin files and third party libraries is very different from deploying non-OSGi Java code. See “Java and OSGi Support” on page 629 and “OSGi Plugin Deployment with IntelliJ IDEA with OSGi Editor” on page 647.

IMPORTANT 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.

Additional Information By Plugin Type

The additional information by plugin type, see the following sections.

Plugin type	For more information
Gosu	<ul style="list-style-type: none">“Example Gosu Plugin” on page 129
Java (no OSGi)	<ul style="list-style-type: none">“Special Notes For Java Plugins” on page 130“Useful Java Plugin APIs” on page 131“Java and OSGi Support” on page 629“Calling Java from Gosu” on page 123 in the <i>Gosu Reference Guide</i>
OSGi (Java with OSGi)	<ul style="list-style-type: none">“Java and OSGi Support” on page 629“Accessing Entity and Typecode Data in Java” on page 633“OSGi Plugin Deployment with IntelliJ IDEA with OSGi Editor” on page 647

Error Handling in Plugins

Where possible, PolicyCenter tries to respond appropriately to errors during a plugin call and performs a default action in some cases.

However, in cases such as policy number generation (`IPolicyNumGenPlugin`), there is no meaningful default behavior, so the action that triggered the plugin fails and displays an error message.

There is not one standard way of handling errors in plugins. It is highly dependent on the plugin interface and the context. Check the method signatures to see what exceptions are expected. In some cases, it might be appropriate to return an empty set or error value if the response return value supports it. Contact the Guidewire Customer Technical Support group if you have questions about specific plugin interfaces and methods.

The PolicyCenter server catches any runtime exception and rolls back any related database transaction, and then displays an error in the application user interface.

Temporarily Disabling a Plugin

By default, PolicyCenter calls Java plugins in the Plugins registry at the appropriate time in the application logic. To disable a plugin in the registry temporarily or permanently, navigate to the Plugins Registry editor in Studio for your plugin implementation. To disable the plugin implementation, deselect the **Enabled** checkbox. To enable the plugin again, select the **Enabled** checkbox.

Example Gosu Plugin

The most important thing to remember about implementing a plugin interface in Gosu is that Gosu versions of the interface methods look different from the Java versions in many cases. The most common problem is that if an interface’s method looks like properties, you must implement the interface as a Gosu property. See “Writing Your Plugin Implementation Class” on page 125.

This topic shows a basic Gosu plugin implementation that uses parameters. For more information about plugin parameters, see “Getting Plugin Parameters from the Plugins Registry Editor” on page 131.

The following Gosu example is a simple messaging plugin that uses parameters:

```
uses java.util.Map;
uses java.plugin;

class MyTransport implements MessageTransport, InitializablePlugin {
    private var _servername : String
```

```

// note the empty constructor. If you do provide an empty constructor, the application
// calls it as the plugin instantiates, which is before application calls setParameters
construct() {
}

override function setParameters(parameters: Map<String, String>) {
    // access values in the MAP to get parameters defined in Plugins registry in Studio
    _servername = parameters["ServerName"] as String
}

override function suspend() {}

override function shutdown() {}

override function setDestinationID(id:int) {}

override function resume() {}

override function send(message:entity.Message, transformedPayload:String) {
    print("MESSAGE SEND ===== ${message.Payload} --> ${transformedPayload}")
    message.reportAck()
}
}

```

If Your Gosu Plugin Needs Java Classes and Library Files

If your Gosu class implements the plugin interface but needs to access Java classes or libraries, you must put these files in the right places in the configuration environment. See “Deploying Non-OSGi Java Classes and JARs” on page 646. Note that the plugin directory setting discussed in that section can have the value `Gosu` for code called from Gosu. Alternatively, you can put your classes in the plugin directory called `shared`.

Special Notes For Java Plugins

If you write your plugin in Java, you must regularly regenerate the Java API libraries to compile against them. For a comparison of Gosu and Java for plugin development, see “Overview of PolicyCenter Plugins” on page 124.

It is important to understand some of the special considerations for writing Java code to use within PolicyCenter. For example:

- The way you access and use Guidewire business data entity instances is different between Gosu and Java. They are in different packages. Also, the way you create new entity instances is different in Java compared to Gosu.
- For plugin interface methods, remember that what Gosu exposes as properties (the `PropertyName` property) appear in Java as getter and setter methods (for example, the `getMyPropertyName` method).

For important information about writing Java in PolicyCenter, see “Java and OSGi Support” on page 629. Note that deployment of OSGi plugins is very different from non-OSGi plugin deployment.

Which Libraries to Compile Java Code Against

After changes to the data model, it might be necessary to regenerate Java API libraries. See “Regenerating Integration Libraries and WSDL” on page 31.

PolicyCenter creates the library JAR files at the path:

`PolicyCenter/java-api/lib`

For important information about deploying Java classes and libraries, see “Java and OSGi Support” on page 629 and “Deploying Non-OSGi Java Classes and JARs” on page 646.

For more information about working with Guidewire entities from your Java code, see “Accessing Entity and Typecode Data in Java” on page 633.

Where To Put Java Class and Library Files Needed By Java Plugins

For Java plugins without OSGi, see “Deploying Non-OSGi Java Classes and JARs” on page 646.

For OSGi plugins (Java with OSGi), see “OSGi Plugin Deployment with IntelliJ IDEA with OSGi Editor” on page 647.

Useful Java Plugin APIs

Entity Data from Java

For important information about working with entity data, see “Accessing Entity and Typecode Data in Java” on page 633.

Getting Current User from a Java Plugin

Sometimes it is useful to determine which user triggered a user interface action that triggered a Java plugin method call. Similarly, it is sometimes useful to determine which application user called a SOAP API triggered a Java plugin method call. In both cases, the Java plugin can use the `CurrentUserUtil` utility class.

To get the current user from a Java plugin, use the following code:

```
myCurrentUser = CurrentUserUtil.getCurrentUser().getUser();
```

Translating Typecodes

Typical PolicyCenter implementations need integration code that interfaces with external systems with different typecodes values than in PolicyCenter. PolicyCenter provides a typecode translation system that you can configure with name/value pairs.

Configure the values using an XML file. To modify the typelist conversion configuration file or to convert typecodes from Gosu or Java, see “Mapping Typecodes to External System Codes” on page 93.

Because typecode translation may occur very frequently in Java plugin code, Java plugins can use a utility class called `TypecodeMapperUtil`.

From external systems, the web service `ITypelistAPI` translates typecodes. It has similar methods to the Java API, such as `getInternalCodeByAlias`.

For example, to translate a typecode with `TypecodeMapperUtil`, use code such as:

```
TypeKey tk = TypecodeMapperUtil.getInternalCodeByAlias("Contact", "ABC:system1", "ATTORNEY");
```

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.

The following Gosu example demonstrates how to define an actual plugin that uses parameters:

```
uses java.util.Map;
uses java.plugin;
```

```

class MyTransport implements MessageTransport, InitializablePlugin {
    private var _servername : String

    // note the empty constructor. If you do provide an empty constructor, the application
    // calls it as the plugin instantiates, which is before application calls setParameters
    construct() {
    }

    override function setParameters(parameters: Map<String, String>) {
        // access values in the MAP to get parameters defined in Plugins registry in Studio
        _servername = parameters["MyServerName"] as String
    }

    // NEXT, define all your other methods required by the MAIN interface you are implementing...
    override function suspend() {}

    override function shutdown() {}

    override function setDestinationID(id:int) {}

    override function resume() {}

    override function send(message:entity.Message, transformedPayload:String) {
        print("MESSAGE SEND ===== ${message.Payload} --> ${transformedPayload}")
        message.reportAck()
    }
}

```

Getting the Local File Path of the Root Directory

For Gosu plugins and Java plugins, you can access the plugin root directory path by getting a special built-in property from the Map. For the key name for root directory, use the name in the static variable `InitializablePlugin.ROOT_DIR`. This parameter is unavailable from OSGi plugins.

Getting the Local File Path of the Temp Directory

For Gosu plugins and Java plugins, you can access the plugin temporary directory path by getting a special built-in property from the Map. For the key name to get the root directory, use the name in the static variable `InitializablePlugin.TEMP_DIR`. This parameter is unavailable from OSGi plugins.

Writing Plugin Templates For Plugins That Take Template Data

Some plugin interface methods have parameters that directly specify their data as simple objects such as `String` objects. Some plugin methods have parameters of structured data such as a `java.util.Map` objects. Some methods take entities such as `Policy`. Some objects might link to other objects, resulting in a potentially large object graph.

However, some plugin interface methods take a single `String` that it must parse to access important parameters. This text data is *template data* specified as plugin method parameters called `templateData`. Template data is the output of running a Gosu template called a *plugin template*. Plugin templates always have the suffix `.gsm`.

IMPORTANT For plugin templates, the file suffix must be `.gsm`. Do not use `.gst`, which is the normal Gosu template extension (see “Template Overview” on page 353 in the *Gosu Reference Guide*).

For example, the policy number generator plugin (`IPolicyNumGenPlugin`) uses template data as a parameter in its methods.

PolicyCenter passes this potentially-large `String` as a parameter to the plugin for the subset of plugins that use template data. This approach lets PolicyCenter pass the plugin all necessary properties in a large data graph but with minimal data transfer.

For example, plugins or the remote systems that they represent often need to analyze a policy's properties to make submission decisions. Frequently, these properties are not just part of the policy itself. Instead, they may be part of the entire graph of objects connected to the policy. For example, properties from only one policy revision or the insured's contact record (for example, age) could be needed to make decisions.

In this case, a submission's object graph can be very large, but a submission number generator plugin does not need all of the policy object graph's full set of data. In this case, the submission number generator plugin might only need a small subset of data in simple *fieldname=fieldname* pairs. The corresponding plugin template might generate simple text like the following

```
PolicyType=TC_homeowners
PolicyEffectiveDate=1/01/2005
...
```

Then, a plugin method can simply parse the text to access the properties. For plugins written in Java, it is easy to use the standard Java class called `Properties`. It can parse a `String` in that format into name/value pairs from which you can extract information using code such as: `propertiesObject.getProperty(fieldname)`.

For example, this Java code takes the `templateData` parameter encoded in the simple format described earlier, and then extracts the value of the `AddressID` property from it:

```
// Create a Java Properties object
Properties policyProperties = new Properties();

try {
    // extract the template data string and load it to the Properties object
    policyProperties.load(new ByteArrayInputStream(templateData.Bytes));
} catch (java.io.IOException IOE) {
    System.out.println("MyPluginName: bad template data");
}

// Extract properties from the Properties object
String myAddressID = policyProperties.getProperty("AddressID");
```

This Gosu code does a similar thing for a Gosu plugin as a private method within the `Gosu` class:

```
private function loadPropertiesFromTemplateData(templateData : String) : Properties
{
    var props = new Properties();
    try{
        props.load(new ByteArrayInputStream(templateData.getBytes()));
        _logger.info("The properties are : " + props);
    }
    catch (e) {
        e.printStackTrace();
        return null;
    }
    return props;
}
```

You can design any text-based data format you want to pass to the plugin in the `templateData` string. If your data is not very structured, Guidewire recommends the simple *fieldname=fieldname* format demonstrated earlier. In some cases, it may be convenient to generate XML formatted data, which permits hierarchical structure despite being a text format. This is especially useful for communicating to external systems that require XML-formatted data. Whatever text-based format you choose to use, you can modify the associated plugin template to generate the desired XML format.

For each plugin method call that takes a `templateData` parameter, PolicyCenter has a Gosu template file in `PolicyCenter/modules/configuration/config/templates/plugins`. PolicyCenter selects the correct plugin using a naming convention:

```
{interface name}_{entity name}.gsm
```

For example, for an plugin to generate a policy revision with template data, the template in that directory would be named `PluginName_PolicyPeriod.gsm`. Although your implementation might be different, it likely look something like the following:

```
PolicyPeriodNumber=<%= PolicyPeriod.PolicyNumber %>
EffDate=<%= PolicyPeriod.RevisionEffDate %>
```

Each plugin that requires template date for some parameters has a different template for each combination of plugin and entity type. After the template runs, it generates template data that looks like the following:

```
PolicyPeriodNumber=HO-7013958-01
EffDate=2007-02-01
```

After the Gosu engine generates a response using the designated Gosu template, the resulting `String` passes to the plugin as the `templateData` parameter to the plugin method. Again, this is only for plugin interface methods that take a `templateData` parameter.

Plugin Registry APIs

Getting References to Plugins from Gosu

To ask the application for the currently-implemented instance of a plugin, call the `Plugins.get(INTERFACENAME)` static method and pass the plugin interface name as an argument. It returns a reference to the plugin implementation. The return result is properly statically typed so you can directly call methods on the result.

For example:

```
uses gw.plugin.Plugins

// Gosu uses type inference to know the type of the variable is IContactSystemPlugin
var plugin = Plugins.get( IContactSystemPlugin )

try{
    plugin.retrieveContact("abc:123" )
}catch(e){
    e.printStackTrace()
    throw new DisplayableException(e.Message)
}
```

Alternatively, you can request a plugin by the plugin name in the Plugins registry. This is important if there is more than one plugin implementation of the interface. For example, this is common in messaging plugins. To do this, use an alternative method signature of the get method that takes a `String` for the plugin name as defined in the Studio plugin configuration.

For example, if your plugin was called `MyPluginName` and the interface name is `IStartablePlugin`:

```
uses gw.plugin.Plugins

// you must downcast to the plugin interface
var contactSystem = Plugins.get( "MyContactPluginRegistryName" ) as IContactSystemPlugin

try{
    contactSystem.retrieveContact("abc:123")
}catch(e){
    e.printStackTrace()
    throw new DisplayableException(e.Message)
}
```

Check Is Enabled

From Gosu you can use the `Plugins` class to determine if a plugin is enabled in Studio. Call the `isEnabled` method on the `Plugins` class and pass either of the following as a `String` value:

- the interface name type (with no package)
- the plugin implementation name in the Plugins registry

For example:

```
uses gw.plugin.Plugins

var contactSystemEnabled = Plugins.isEnabled( IContactSystemPlugin )
```

Getting References to Java Plugins from Java

From Java you can get references to other installed plugins either by class or by the string representation of the class. Do this using methods on the `PluginRegistry` class within the `guidewire.pl.plugin` package. Get the plugin by class by calling the `getPlugin(Class)` method. Get the plugin by name calling the `getPluginByName(String)` method.

This 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.

Also, this allows you 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.

IMPORTANT Do not use this API from Gosu. From Gosu, call the `Plugins.get(INTERFACENAME)` static method and pass the plugin interface name or plugin name as an argument. See earlier in this topic for details.

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. Messaging plugins instantiate only on the batch server. The other non-batch servers have zero instances of message request, message transport, and message reply plugins. See “[Messaging Flow Details](#)” on page 295. 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 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 must support multiple simultaneous calls to the plugin in general. For example, PolicyCenter might call two different plugin methods at the same time. You must ensure multiple method 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).

Collectively, these requirements describe thread safety. You must ensure your implementation is thread safe.

IMPORTANT For important information about concurrency, see “[Concurrency](#)” on page 375 in the *Gosu Reference Guide*.

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 a 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 (see “Design Plugin Implementations to Support Server Clusters” on page 139). 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 (but not natively in Gosu) that locks access between threads to shared resources such as static variables.

WARNING Avoid static variables in plugins if at all possible. PolicyCenter may call plugins from multiple process threads and in some cases this could be dangerous and unreliable. Additionally, this type of problem is extremely difficult to diagnose and debug.

For important information about concurrency, see “Concurrency” on page 375 in the *Gosu Reference Guide*.

For more information about concurrency and related APIs in Java, see:

<http://java.sun.com/docs/books/tutorial/essential/concurrency/index.html>

The following sections list some common approaches for thread safety with static variable in Java:

- “Using Java Concurrent Data Types, Even from Gosu” on page 137
- “Using Synchronized Methods (Java Only)” on page 137
- “Using Java Synchronized Blocks of Code (Java only)” on page 138

For thread safety issues in Gosu, see “Gosu Static Variables and Gosu Thread Safety” on page 136.

Additionally, note some similar issues related to multi-server (cluster) plugin design in “Design Plugin Implementations to Support Server Clusters” on page 139.

Gosu Static Variables and Gosu Thread Safety

The challenges of static variables and thread safety applies to Gosu classes, not just Java. This affects Gosu in plugin code and also for Gosu classes triggered from rules sets. The most important thing to know is that static variables present special challenges to ensure your code is thread safe.

The typical way to create a Gosu class with static variable is with code like:

```
class MyClass {  
    static var _property1 : String;  
}
```

You must be as careful in Gosu with static variables and synchronizing data in them to be thread safe as in Java. Use the Java concurrent data types describes in this section to ensure safe access.

WARNING Thread safety APIs that use blocking can affect performance negatively. For highest performance, use such APIs wisely and test your code under heavy loads that test the concurrency.

For important information about concurrency, see “Concurrency” on page 375 in the *Gosu Reference Guide*.

Using Java Concurrent Data Types, Even from Gosu

The simplest way to 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.

WARNING 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.

For important information about concurrency, see “Concurrency” on page 375 in the *Gosu Reference Guide*.

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. For more information about plugin parameters, see “Special Notes For Java Plugins” on page 130.

On a related note, Java class constructors cannot be synchronized; using the Java keyword `synchronized` with a constructor generates a syntax error. Synchronizing constructors does not make sense because only the thread that creates an object has access to it during the time Java is constructing it.

For important information about concurrency, see “Concurrency” on page 375 in the *Gosu Reference Guide*.

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, this 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. See “Using Synchronized Methods (Java Only)” on page 137.

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.

WARNING Thread safety APIs that use blocking can affect performance negatively. For highest performance, use such APIs wisely and test your code under heavy loads that test the concurrency.

For important information about concurrency, see “Concurrency” on page 375 in the *Gosu Reference Guide*.

Avoid Singletons Due to Thread-Safety Issues

The thread safety problems discussed in the previous section apply to any Java object that has only a single instance (also referred to as a *singleton*) implemented using static variables. Because static variable access 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.

This is 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;  
    }  
}
```

For more information about singletons in Java, see:

<http://java.sun.com/developer/technicalArticles/Programming/singletons>

If you absolutely must use a singleton, you must synchronize updates to class static variables as discussed at the beginning of “Plugin Thread Safety” on page 135.

WARNING Avoid creating singletons, which are classes that enforce only a single instance of the class. If you really must use singletons, you must use the synchronization techniques discussed in “Plugin Thread Safety” on page 135 to be thread safe.

For important other information about concurrency, see “Concurrency” on page 375 in the *Gosu Reference Guide*.

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.

Note: There is an exception for this cluster rule: messaging plugins exist only for the single server designated the batch server. See “Messaging Flow Details” on page 295.

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.

The thread safety synchronization techniques in “Plugin Thread Safety” on page 135 are insufficient to synchronize data shared across multiple servers in a cluster. Each server has its own Java virtual machine, so it has its own data space. 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.

You must implement your own approach to ensure access to shared resources safely even if accessed simultaneously by multiple threads and on multiple servers.

For important information about concurrency, see “Concurrency” on page 375 in the *Gosu Reference Guide*.

Reading System Properties in Plugins

You might want to test plugins in multiple deployment environments without recompiling plugins. For example, perhaps if a plugin runs on a test server, then the plugin queries a test database. If it runs on a production server, then the plugin queries a production database.

Or, you might want a plugin that can be run on multiple machines within a cluster with each machine knowing its identity. You might want to implement unique behavior within the cluster. Alternatively, add this information to log files on the local machine and in the external system also.

For 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-line 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 Plugins Registry editor in Studio.

Gosu plugins can query system properties using the methods `getEnv`, `getServerId` and `isBatchServer`, all on the `ServerUtil` class. For example the following Gosu expression evaluate to the current value of the `env` system property, the server ID, and whether this server is the batch server:

```
var envValue = gw.api.system.server.ServerUtil.getEnv("gw.pc.env")
var serverID = gw.api.system.server.ServerUtil.getServerId()
var isBatchServer = gw.api.system.server.ServerUtil.isBatchServer()
```

Java plugins can query system properties using code such as the following example, which gets the `env` property:

```
java.lang.System.getProperty("gw.pc.env");
```

See also

- “Clustering Application Servers” on page 77 in the *System Administration Guide*.
- For more information about `env` and `serverId` settings, see “Specifying Environment Properties in the `<registry>` Element” on page 15 in the *System Administration Guide*.
- “Using the Plugins Registry Editor” on page 131 in the *Configuration Guide*.

Do Not Call Local Web Services From Plugins

Do not call locally-hosted web service APIs (SOAP 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.

If you have questions about how to refactor your code to avoid local loopback API calls over web services, contact Guidewire Customer Support.

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.

These methods take two parameters:

- An initial value for the sequence, if it does not yet exist.
- A `String` with up to 256 characters that uniquely identifies the sequence. This is the sequence key (`sequenceKey`).

For example, suppose you want to get a new number from a sequence called `WidgetNumber`.

In Gosu, use code like the following:

```
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 1. If other code calls this method again, the return value is two, three, or some larger number depending on how many times any code requested numbers for this sequence key.

Similarly, in Java, use the code:

```
nextNum = gw.api.system.database.SequenceUtil.next(10, "WidgetNumber");
```

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 (non-product) 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.

For example, Apache Tomcat provides a web application called Manager that provides this functionality. For documentation on this Apache Tomcat Manager, refer to:

<http://jakarta.apache.org/tomcat/tomcat-4.1-doc/manager-howto.html>

Summary of All PolicyCenter Plugins

The following table summarizes the plugin interfaces that PolicyCenter defines. For a table that summarizes the plugin interfaces that ContactManager defines, see “ContactManager Plugins” on page 255 in the *Contact Management Guide*.

PolicyCenter Plugin Interface	Description	Maximum enabled implementations
Contact-related plugins		
IContactConfigPlugin	Configuration of contacts, such as what address contact roles are available, and getting the display name for an account contact role type. See “Configuring How PolicyCenter Handles Contacts” on page 531.	1
IContactSystemPlugin	Search for contacts and retrieve contacts from an external system. For more information. See “Integrating with a Contact Management System” on page 525.	1
IAccountSyncable	Customize how PolicyCenter synchronizes contact with accounts. For more information. See “Synchronizing Contacts with Accounts” on page 532.	1
IAccountContactPlugin	Configure how to copy properties between account contacts. See “Account Contact Plugin” on page 533.	1
IAccountContactRolePlugin	Configure how to copy properties between account contact roles and copy pending updates. See “Account Contact Role Plugin” on page 533.	1
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 255.	1
IGeocodePlugin	Geocoding support within PolicyCenter. See “Geographic Data Integration” on page 229.	1
IContactSearchAdapter	<i>Internal only. Never use or implement this plugin.</i>	-
IAddressBookAdapter	<i>Internal only. Never use or implement this plugin.</i>	-

PolicyCenter Plugin Interface	Description	Maximum enabled implementations
Other PolicyCenter plugins		
AccountLocationPlugin	For account locations and policy locations, customize how PolicyCenter: <ul style="list-style-type: none">• Determines that two locations are the same• Clones a location See “Location Plugin” on page 157.	1
ConversionOnRenewal	Configures actions to perform if a renewal fails during converting. See “Conversion on Renewal Plugin” on page 178.	1
IAccountPlugin	Manipulates accounts. See “Account Plugin Details” on page 151.	1
IArchivingSource	Stores and retrieves archived policies from an external backing source. See “Archiving Integration” on page 557.	1
IAuditSchedulePatternSelectorPlugin	Customize how PolicyCenter chooses the audit schedule pattern for cancellation and expiration. See “Audit Schedule Selector Plugin” on page 168.	1
IBillingSummaryPlugin	Generates billing summaries for the billing summary screen. See “Implementing the Billing Summary Plugin” on page 500	1
IBillingSystemPlugin	Interacts with an external billing system for actions that PolicyCenter initiates. See “Implementing the Billing System Plugin” on page 490	1
IClaimSearchPlugin	Searches for claims. See “Claim Search from PolicyCenter” on page 507.	1
IEffectiveTimePlugin	Determines the time of day (since midnight) for a job. See “Effective Time Plugin” on page 161.	1
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 177.	1
IExchangeRateSet	<i>Internal only. Never use or implement this plugin.</i>	-
IFXRatePlugin	Handles exchange rate conversion. See “Implementing an Exchange Rate Service” on page 607 in the <i>Configuration Guide</i> .	1
IImpactTestingPlugin	<i>Only for customers who license Guidewire Rating Management.</i> This plugin configures rating routine impact testing. See “Impact Testing Plugin” on page 577 in the <i>Configuration Guide</i> .	1
IJobCreation	Creates a job process. See “Job Process Creation Plugin” on page 152	1
IJobProcessCreationPlugin	Creates the right job process subtype. See “Job Process Creation Plugin” on page 152.	1
ILossHistoryPlugin	Handles loss history summaries for a policy. See “Loss History Plugin” on page 157.	1
IMotorVehicleRecordPlugin	Generates a request to the motor vehicle record (MVR) provider and returns the MVR data. See “Motor Vehicle Record (MVR) Plugin” on page 171.	1
INotificationPlugin	Customizes the minimum lead time and the maximum lead time for different types of notifications. See “Notification Plugin” on page 166.	1
IPolicyEvaluationPlugin	Customizes how PolicyCenter adds underwriter issues (<code>UWIssue</code> objects) on a policy period. See “Policy Evaluation Plugin” on page 164.	1

PolicyCenter Plugin Interface	Description	Maximum enabled implementations
IPolicyNumGenPlugin	Generates policy numbers of two types: core policy numbers and current policy revision numbers. See "Policy Number Generator Plugin" on page 150.	1
IPolicyPaymentPlugin	Returns the filtered reporting plans based on the policy. See "Policy Payment Plugin" on page 162.	1
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 Customization" on page 431	1
IPolicyPeriodPlugin	<p>Various important policy period customizations, including the following:</p> <ul style="list-style-type: none"> • calculating the period end date from a TermType • calculating a TermType 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 PolicyPeriod • 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 154.</p>	1
IPolicyPlugin	Lets PolicyCenter know whether it is OK to start various jobs, based on dynamic calculations on the policy. See "Policy Plugin" on page 158.	1
IPolicyTermPlugin	Calculates values related to policy terms. See "Policy Term Plugin" on page 153	1
IRateRoutinePlugin	<i>Only for customers who license Guidewire Rating Management.</i> This plugin configures processing of rate routines. See "Rate Routine Plugin" on page 573 in the <i>Configuration Guide</i> .	1
IRatingPlugin	This is the main plugin interface that defines the interaction between the application and a rating engine. See "Rating Integration" on page 353.	1
IReferenceDatePlugin	Evaluates the type of date to use to calculate the reference date on a given object. See "Reference Date Plugin" on page 168.	1
IRenewalPlugin	Handles renewals. See "Renewal Plugin" on page 165.	1
ITerritoryCodePlugin	Allows an external system to provide information about territories. See "Territory Code Plugin" on page 251.	1
IUWCompanyPlugin	Finds all the underwriting companies that are available for the states in the set. See "Underwriting Company Plugin" on page 163.	1
IVinPlugin	Allows an external system to provide information about vehicles. See "Vehicle Identification Number Plugin" on page 253.	1
JobNumberGenPlugin	Generates a new, unique job number. See "Policy Number Generator Plugin" on page 150.	1
PCPurgePlugin	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 173.	1

PolicyCenter Plugin Interface	Description	Maximum enabled implementations
ProrationPlugin	Prorate 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 169.	1
QuotingDataPlugin	Provides methods for saving quoting data to an external database. See “Quoting Data Plugin” on page 595 in the <i>Configuration Guide</i> .	1
WorksheetExtractPlugin	<i>Only for customers who license Guidewire Rating Management.</i> This plugin identifies which rating worksheet to extract and extracts the worksheet data to files. See “Worksheet Extract Plugin” on page 569 in the <i>Configuration Guide</i> .	1
WorksheetPurgePlugin	<i>Only for customers who license Guidewire Rating Management.</i> This plugin configures how PolicyCenter purges rating worksheets. See “Worksheet Purge Plugin” on page 569 in the <i>Configuration Guide</i> .	1
Authentication plugins (see “Authentication Integration” on page 181)		
AuthenticationServicePlugin	The authentication service plugin authorizes a user from a remote authentication source, such as a corporate LDAP or other single-source sign-on system.	1
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.	1
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.	1
DBAuthenticationPlugin	Allows you 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.	1
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 59.	1
Document content and metadata plugins (see “Document Management” on page 191)		
IDocumentContentSource	Provides access to a remote repository of documents, for example to retrieve a document, store a document, or remove a document from a remote repository. PolicyCenter implements this with a default version, but for maximum data integrity and feature set, implement this plugin and link it to a commercial document management system.	1

PolicyCenter Plugin Interface	Description	Maximum enabled implementations
IDocumentMetadataSource	Stores metadata associated with a document, typically to store it in a remote document management system; see IDocumentContentSource mentioned earlier for a related plugin. By default, PolicyCenter stores the metadata locally in the PolicyCenter database if you do not define it. For maximum data integrity and feature set, implement this plugin and link it to a commercial document management system.	1
Document production plugins (see “Document Production” on page 205)		
IDocumentProduction	Generates documents from a template. For example, from a Gosu template or a Microsoft Word template. This plugin can create documents synchronously and/or asynchronously. Note: Only register one implementation of this plugin. However, there are multiple other classes that implement this interface and handle one document type. The registered implementation of this plugin interface dispatches requests to the other classes that implement this same interface. See “Document Production” on page 205.	1 (see note)
IDocumentTemplateSource	Provides access to a repository of document templates that can generate forms and letters, or other merged documents. An implementation may simply store templates in a local repository. A more sophisticated implementation might interface with a remote document management system.	1
IEmailTemplateSource	Gets email templates. By default, PolicyCenter stores the email templates on the server but you can customize this behavior by implementing this plugin.	1
IDocumentTemplateSerializer	<i>Use the built-in version of this plugin using the “Plugins registry” but generally it is best not implement your own version.</i> 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.	1
Messaging plugins (see “Messaging and Events” on page 289)		
MessageTransport	This is the main messaging plugin interface. This plugin sends a message to an external/remote system using any transport protocol. This could involve submitting to a messaging queue, calling a remote API call, saving to special files in the file system, sending e-mails, or anything else you require. Optionally, this plugin can also acknowledge the message if it is capable of sending synchronously (that is, as part of a synchronous send request). 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 <i>plugin name</i> . Use the plugin name when you configure the messaging destination in the separate Messaging editor in Studio.	Multiple
MessageRequest	Optional pre-processing of messages, and optional post-send-processing (separate from post-acknowledgement processing). 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 <i>plugin name</i> . Use the plugin name when you configure the messaging destination in the separate Messaging editor in Studio.	Multiple

PolicyCenter Plugin Interface	Description	Maximum enabled implementations
MessageReply	<p>Handles asynchronous acknowledgements of a message. After submitting an acknowledgement 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.</p> <p>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 <i>plugin name</i>. Use the plugin name when you configure the messaging destination in the separate Messaging editor in Studio.</p>	Multiple
Other plugins		
InboundIntegrationStartablePlugin	<p>High performance inbound integrations, with support for multi-threaded processing of work items. See “Multi-threaded Inbound Integration” on page 267</p>	Multiple
ITestingClock	<p>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 <i>for development (non-production) use only</i>. It programmatically changes the system time to accelerate the perceived passing of time within PolicyCenter.</p>	1
IAddressAutocompletePlugin	<p>Configures how address automatic completion and fill-in operate. See “Automatic Address Completion and Fill-in Plugin” on page 253.</p>	1
IPhoneNormalizerPlugin	<p>Normalizes phone numbers that users enter through the application and that enter the database through data import. See “Phone Number Normalizer Plugin” on page 254.</p>	1
IEncryptionPlugin	<p>Encodes or decodes a String 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 String or reversing that process. The built-in implementation of this plugin does nothing. See “Encryption Integration” on page 239.</p>	Multiple
	<p>The IEncryption plugin interface supports multiple implementations to support changes to encryption algorithms. See “Changing Your Encryption Algorithm Later” on page 244.</p>	
IBaseURLBuilder	<p>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 256.</p>	1
ManagementPlugin	<p>The external management interface for PolicyCenter, which allows you to implement management systems such as JMX, SNMP, and so on. See “Management Integration” on page 247.</p>	1

PolicyCenter Plugin Interface	Description	Maximum enabled implementations
IScriptHost	<p><i>You can use the built-in version of this plugin using the Plugins registry APIs but do not attempt to implement your own version.</i></p> <p>Get the current implementation using the Plugins registry and use it to evaluate Gosu code to provide context-sensitive data to other services. For example, property data paths defined with Gosu expressions. For example, this service could evaluate a String that has the Gosu expression "policy.myFieldName" at run time.</p>	1
IProcessesPlugin	<p>Instantiates custom batch processing classes so they can be run on a schedule or on demand. See "Implementing the Processes Plugin" on page 591.</p>	1
IStartablePlugin	<p>Creates new plugins that immediately instantiate and run on server startup. You can register multiple startable plugin implementations for different tasks. See "Startable Plugins Overview" on page 259.</p>	Multiple
IPreupdateHandler	<p>Implements your preupdate handling in plugin code rather than in the built-in rules engine. See "Preupdate Handler Plugin" on page 256. In PolicyCenter, all preupdate rule sets in the base configuration are deprecated, so this is the replacement approach.</p>	1
IWorkItemPriorityPlugin	<p>Calculates the processing priority of a work item. See "Work Item Priority Plugin" on page 255.</p>	1
IActivityEscalationPlugin	<p>Overrides the behavior of activity escalation instead of simply calling rule sets. See "Exception and Escalation Plugins" on page 257.</p>	1
IGroupExceptionPlugin	<p>Overrides the behavior of group exceptions instead of simply calling rule sets. See "Exception and Escalation Plugins" on page 257.</p>	1
IUserExceptionPlugin	<p>Overrides the behavior of user exceptions instead of simply calling rule sets. See "Exception and Escalation Plugins" on page 257.</p>	1
ClusterBroadcastTransportFactory ClusterFastBroadcastTransportFactory ClusterUnicastTransportFactory	<p><i>For internal use only.</i> A plugin implementation is registered in the Plugins registry in the default configuration for these plugin interfaces. However, they are unsupported for customer use. Do not change settings in the Plugins registry for these interfaces. Do not use or implement these plugin interfaces.</p>	n/a

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.

This topic includes:

- “Policy Number Generator Plugin” on page 150
- “Account Plugin Details” on page 151
- “Job Process Creation Plugin” on page 152
- “Job Number Generator Plugin” on page 152
- “Policy Term Plugin” on page 153
- “Policy Period Plugin” on page 154
- “Loss History Plugin” on page 157
- “Location Plugin” on page 157
- “Policy Plugin” on page 158
- “Effective Time Plugin” on page 161
- “Policy Payment Plugin” on page 162
- “Underwriting Company Plugin” on page 163
- “Policy Evaluation Plugin” on page 164
- “Renewal Plugin” on page 165
- “Notification Plugin” on page 166
- “Reference Date Plugin” on page 168
- “Audit Schedule Selector Plugin” on page 168
- “Proration Plugin” on page 169
- “Motor Vehicle Record (MVR) Plugin” on page 171
- “Policy Hold Job Evaluation Plugin” on page 173
- “Quote Purging Plugin” on page 173
- “ETL Product Model Loader Plugin” on page 177

- “Conversion on Renewal Plugin” on page 178

See also

- For general information about plugins, see “Plugin Overview” on page 123.
- For the complete list of PolicyCenter plugins, see “Summary of All PolicyCenter Plugins” on page 141.

Policy Number Generator Plugin

The policy number generator plugin interface (`IPolicyNumGenPlugin`) is responsible for generating a policy number. The plugin interface has one method. PolicyCenter only calls this method after creating a new contractual period. The number it generates represents that contractual period. For example, PolicyCenter calls this while creating a branch in a new contractual period for a submission or a renewal job.

Note: There is no policy number that represents all contractual periods on one policy.

If the user attempts a rewrite job, the user interface lets you choose whether to reuse the policy number for that contractual period or whether to generate a new one. The user interface sets the value with `Rewrite.isChangePolicyNumber()`, and if it is true then PolicyCenter calls this plugin to generate a new number.

The plugin interface method is `genNewPeriodPolicyNumber`. This method takes a `PolicyPeriod` entity and must return a `String` containing the policy number. If you cannot generate a policy number, throw a `RemoteException` exception. For example, if you cannot access the external system that generates the policy number, throw a `RemoteException` exception.

The `PolicyNumGenPlugin` plugin is the built-in implementation of this interface. This plugin generates a random string of 10 digits, and does not check for duplicates in the database.

IMPORTANT The `PolicyNumGenPlugin` plugin implementation 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.

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.

WARNING 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.

See also

- For an example implementation of this plugin in Java, see “Special Notes For Java Plugins” on page 130.
- “Field Validators” on page 277 in the *Configuration Guide*.

Account Plugin Details

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 carrier determines whether this client already has insurance with the carrier 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.

Customizing 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.

Generate 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. You might want to use the sequence APIs to do this, discussed in “Creating Unique Numbers in a Sequence” on page 140.

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.

Customizing 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.

Customizing 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 details, see “Account Web Services” on page 102.

Job Process Creation Plugin

The job process creation plugin (`JobProcessCreationPlugin`) 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.

This 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

- “Gosu Classes for Jobs” on page 614 in the *Configuration Guide*

Job Number Generator Plugin

The job number generator (`IJobNumberGenPlugin`) plugin interface is responsible for generating a job number.

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 as follows:

- 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<random 5>`, 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.

IMPORTANT The `JobNumberGenPlugin` plugin implementation 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.

Policy Term Plugin

The policy term plugin (`IPolicyTermPlugin`) calculates values related to policy terms. PolicyCenter includes a built-in implementation of this plugin (`PolicyTermPlugin`). Find the built-in implementation in Studio in the `gw.plugin.policyterm.impl` package. You can modify this plugin to customize it. Or, implement your own instance of this plugin to customize PolicyCenter application logic.

Calculating the Period End Date from a Term Type

Implement the `calculatePeriodEnd` method which returns the default expiration date.

The method signature is:

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

The input parameters to `calculatePeriodEnd` are:

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

This plugin method sets 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	Then 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	Then 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	Then subsequent policy period end dates are the last day of the month.

For example, the month of August has 31 days. The first half-year term of a policy starts on August 30, 2014. Date reconciliation ensures two half-year terms provide coverage for a full year:

Period start	Period end	Description
August 30, 2014	February 28, 2015	Meets condition 2. The period end month, February, has fewer than 30 days, so the period ends on the last day of February.
February 28, 2015	August 30, 2015	Conditions 2 and 3 do not apply, therefore it meets condition 1. 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 `PolicyPeriod.Job.shouldPerformDateReconciliation` flag to determine whether to reconcile the date. By default, the flag is `false`. However, the `SubmissionImpl` and `RenewalImpl` classes set the flag to `true`.

Calculating 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
}
```

Policy Period Plugin

The policy period plugin (`IPolicyPeriodPlugin`) is responsible for a variety of tasks related to `PolicyPeriod` entities. PolicyCenter includes a built-in implementation of this plugin. Find the built-in implementation in Studio in the `gw.plugin.policyperiod.impl` package. You can modify this plugin to customize it. You can also implement your own instance of this plugin to customize PolicyCenter application logic.

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 policy period plugin implementation (`gw.plugin.policyperiod.impl.PolicyPeriodPlugin`) 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`. 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 means the date on which for accounting purposes the premium is considered as written.

The method must return 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.

Customizing Behavior 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.

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 or reinstatement.

For example, you might want to include special custom entities that you use for integration or accounting reasons. You might not, however, 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 must return a set of entity types (`IEntityType`).

For example, you can define the set directly in-line using Gosu shorthand for set creation:

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

Customizing Behavior 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` (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 `WCPolicyLineMethod` 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.

Customizing Behavior 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 `WCPolicyLineMethod` class has a `postCreateDraftBranchInNewPeriod` method. This method updates workers' compensation jurisdiction rating period start dates (RPSDs) after creating a draft in the new period.

Customizing Behavior Before Promoting a Branch

Implement the `prePromote` method to customize behavior before a branch is promoted. In other words, 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. For more information about branch structure and branch promotion, see "Policy Revisioning" on page 489 in the *Application Guide*.

In general, there is nothing PolicyCenter requires you to do at this point but you can customize the logic as necessary.

Customizing Behavior After Handling a Preemption

Implement the `postHandlePreemption` method to customize behavior after a preemption is handled. Preemption is name for the situation in which two concurrent jobs for a policy were based on the same branch. After the second one 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. For more on this topic, see "Preempted Jobs" on page 513 in the *Application Guide*.

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.

Loss History Plugin

The loss history plugin (`ILossHistoryPlugin`) interface retrieves the loss history for an account or a policy from an external system.

PolicyCenter includes a built-in implementation of this plugin, which queries the Guidewire PolicyCenter product for policy data. However, you can override this plugin to return this information from another external system of record for the loss history data.

The methods for this plugin are:

- `getLossFinancialsForAccount` – Gets the loss financials for an account (identified by its account ID string) for the most recent policy periods of all policies on the account. Returns an array of `LossFinancials` objects.
- `getLossFinancialsForPolicy` – Returns the loss financials info for the policy period as-of the given date (specified as a `java.util.Calendar` object). It returns financials for the entire period effective at that date. Returns an array of `LossFinancials` objects.

Location Plugin

You can customize how PolicyCenter treats account locations and policy locations using the location plugin (`AccountLocationPlugin`). The built-in implementation of this plugin is `AccountLocationPluginImpl`.

The basic tasks you can customize are:

- 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.

See also

- “Methods to Remove a Location from a Policy Line” on page 207 in the *Product Model Guide*

Comparing 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.

Customizing 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.

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.

WARNING 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.

The method signature is:

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

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.

Cancellation Starting

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:

- 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 cancelling 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.

Reinstatement Starting

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:

- 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.

Rewrite Starting

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:

- 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.

Policy Change Starting

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:

- 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.

IMPORTANT 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.

Audit Starting

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:

- 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.

Issuance Starting

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:

- 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.

Renewal Starting

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:

- 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 Built-in Plugin Implementation

The built-in implementation 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.

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:

Method	Description
<code>getSubmissionEffectiveTime</code>	Gets effective time for a submission job.
<code>getSubmissionExpirationTime</code>	Gets expiration time for a submission job. Includes an argument the effective date of the submission (including the time component) as returned by <code>getSubmissionEffectiveTime</code> .
<code>getPolicyChangeEffectiveTime</code>	Gets effective time for a policy change job.
<code>getCancellationEffectiveTime</code>	Gets effective time for a cancellation job.

Method	Description
getReinstatementEffectiveTime	Gets effective time for a reinstatement job.
getReinstatementExpirationTime	Gets expiration time for a reinstatement job. Includes an argument for the effective date of the reinstatement (including the time component as returned by getReinstatementEffectiveTime).
getRewriteEffectiveTime	Gets effective time for a rewrite job.
getRewriteExpirationTime	Gets expiration time for a rewrite job. Includes an argument for the effective date of the rewrite (including the time component as returned by getRewriteEffectiveTime).
getRenewalEffectiveTime	Gets effective time for a renewal job.
getRenewalExpirationTime	Gets expiration time for a renewal job. Includes an argument the effective date of the renewal (including the time component as returned by getRenewalEffectiveTime).

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

Reimplement or customize this plugin to provide different logic.

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](#).

This plugin has only a single method, which is called `filterReportingPlans`. This method takes a `Policy` entity and an array of `PaymentPlanSummary` objects. Your method must return an array of payment plan summary objects (`PaymentPlanSummary`), which collectively represents the subset of plans that are available for the policy.

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

The following example implementation filters payment plans. If the policy is a workers' compensation policy, the method just returns the input payment plans. Otherwise, the method generates a list of all plans with a `PaymentCode` property set to `null`.

```
class PolicyPaymentPlugin implements IPolicyPaymentPlugin {  
  
    override function filterReportingPlans( policy : Policy, plans : PaymentPlanSummary[] )  
        : PaymentPlanSummary[]  
    {  
        if (policy.ProductCode.equals("WorkersComp")) {  
            return plans  
        }  
        else {  
            return plans.where( \ p -> p.PaymentCode == null )  
        }  
    }  
}
```

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

Underwriting Company Plugin

To customize how PolicyCenter finds which underwriting companies are available for a set of states, 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 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 states 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`. Use the `allStates` parameter as follows:

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

The built-in implementation of this plugin calls a built-in class to find underwriting companies by states 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()
}
}

```

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` type-list. 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 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.

This 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

- “Creating Underwriting Issues” on page 498 in the *Configuration Guide*
- “Configuring Underwriting Authority” on page 493 in the *Configuration Guide*

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. By default, this process runs daily at 1:00 AM. For more information about work queues, see:

- “Work Queues” on page 112 in the *System Administration Guide*
- “Scheduling Work Queue Writers and Batch Processes” on page 117 in the *System Administration Guide*
- “Scheduling Work Queue Writers and Batch Processes” on page 117 in the *System Administration Guide*

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.

IMPORTANT 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.

For the built-in implementation:

- 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.

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.

Note: This topic discusses *renewal offers*. There is important documentation about how renewal offers work in the billing integration documentation in the subtopic “Billing Implications of Renewals or Rewrites” on page 476.

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.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`). For more information about the notification configuration system table, see “Notification Config System Table” on page 70 in the *Product Model Guide*.

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.

There are four total required methods in this plugin:

- “Notification by Action Type (Minimum)” on page 167
- “Notification by Action Type (Maximum)” on page 167
- “Notification by Category (Minimum)” on page 167
- “Notification by Category (Maximum)” on page 167

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 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:

- `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. The method signature is:

```
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.

This 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

This 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. The method signature is:

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

Reference Date Plugin

PolicyCenter calls the currently-registered reference date plugin (`IReferenceDatePlugin`) 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` in the package `gw.plugin.productmodel.impl`.

You must implement methods to get several types of dates as follows:

- “Coverage Reference Date” on page 168
- “Exclusion Reference Date” on page 168
- “Modifier Reference Date” on page 168
- “Period Reference Date” on page 168

All these methods must return 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 takes as parameters:

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

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

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")  
    }  
}
```

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 this plugin (`ProrationPlugin`) 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.

To add a plugin parameter to the proration plugin

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

See also

- “`IgnoreLeapDayForEffDatedCalc`” on page 69 in the *Configuration Guide*.

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: <ul style="list-style-type: none"> • a period start date • a period end date • a date to prorate to • an amount as a <code>BigDecimal</code> Return the new amount as a <code>BigDecimal</code> value.
<code>scaleAmount</code>	Optional	Scales an amount with the rounding level and rounding mode passed to the constructor. There are two method signatures for this method: <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. 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
<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. See below for details.
<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. See below for details.

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:

- The `IgnoreLeapDays` parameter on the plugin interface is `true`.
- The `IgnoreLeapDays` parameter is not specified, and the `config.xml` parameter `IgnoreLeapDayForEffDatedCalc` parameter 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 358.

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 then open `IMotorVehicleRecordPlugin.gwp`.

Overview of Motor Vehicle Record Integration

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.

How PolicyCenter Uses Motor Vehicle Records

A carrier 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

- “Motor Vehicle Record Object Model” on page 278 in the *Application Guide*
- “System Table to Set Parameters for Retrieving Personal Auto Motor Vehicle Records” on page 282 in the *Application Guide*

Implementations of the Motor Vehicle Record Plugin

The base configuration of PolicyCenter provides a built-in demonstration implementation of the Motor Vehicle Record plugin, `DemoMotorVehicleRecordPlugin.gos`. 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 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.motorvehiclerecord` 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 accepts an array of `IMVRSUBJECT` instances as its sole input parameter.

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:

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, 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 this 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 a active blocking policy hold
- Jobs not evaluated since the last time the policy hold changed

To evaluate a job against a policy hold, implement this 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.

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.gs`. To edit the registry for `PurgePlugin`, in Studio, navigate to `configuration → config → Plugins → registry`, and then open `PurgePlugin.gwp`.

See also

- “Quote Purging” on page 449 in the *Application Guide*
- “Configuring Quote Purging” on page 477 in the *Configuration Guide*
- “Quote Purging Batch Processes” on page 481 in the *Configuration Guide*
- “Quote Purging Configuration Parameters” on page 74 in the *Configuration Guide*

Quote Purging Plugin Methods and Getters

This topic describes the methods and property getters in the `PurgePlugin` plugin interface.

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.

See also

- “Purge Context Object” on page 177

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:

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

IMPORTANT Guidewire does not support configuring quote purging to remove archived policy periods. Do not attempt to implement this functionality without first contacting Guidewire Customer Support for important guidance.

See also

- “Purge Context Object” on page 177

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 as writing to the log file a message that the purge completed.

See also

- “Purge Context Object” on page 177

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. This 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
<code>NoActionRequired</code>	This status indicates that no purging is necessary. The code simply returns <code>null</code> .
<code>Pruned</code>	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.
<code>Unknown</code>	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.

In the base configuration, the `PCPurgePlugin` class does the following in the `getPurgeJobDate` method:

1. Calculates the purge date for closed jobs.
2. 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. For more information, see "Quote Purging Configuration Parameters" on page 74 in the *Configuration Guide*. If `PurgeJobsPolicyTermDaysCheckDisabled` is `true`, then this date is `null`.
3. 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 does the following in the `getPruneJobDate` method:

1. Calculates the prune date for closed jobs.
2. 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. For more information, see "Quote Purging Configuration Parameters" on page 74 in the *Configuration Guide*. If `PruneVersionsPolicyTermDaysCheckDisabled` is `true`, then this date is `null`.
3. 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.

IMPORTANT Guidewire does not support configuring quote purging to remove archived policy periods. Do not attempt to implement this functionality without first contacting Guidewire Customer Support for important guidance.

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:

Property	Description
<code>Policy</code>	Contains one of the following: <ul style="list-style-type: none">The policy being purgedThe parent policy of the job being purgedThe parent policy of the policy period being purged
<code>Job</code>	The job being purged or the parent job of the policy period being purged, if any.
<code>PolicyPeriod</code>	The policy period being purged.
<code>PurgePolicy</code>	Boolean. Value is <code>true</code> if a policy is being purged.
<code>PurgeJob</code>	Boolean. Value is <code>true</code> if a job is being purged.
<code>PrunePeriod</code>	Boolean. Value is <code>true</code> 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`.

IMPORTANT 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.

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 (ETL) 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 Loader” on page 113 in the *Product Model Guide*

ETL Product Model Loader 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`):

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

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`. This class implements the 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.

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.

This topic discusses *plugins*, which are software modules that PolicyCenter calls to perform an action or calculate a result. For information on plugins, see “Plugin Overview” on page 123. For the complete list of all PolicyCenter plugins, see “Summary of All PolicyCenter Plugins” on page 141.

This topic includes:

- “Overview of User Authentication Interfaces” on page 181
- “User Authentication Source Creator Plugin” on page 183
- “User Authentication Service Plugin” on page 185
- “Deploying User Authentication Plugins” on page 188
- “Database Authentication Plugins” on page 189
- “ContactManager Authentication” on page 190

Overview of User Authentication Interfaces

There are several mechanisms to log in to PolicyCenter:

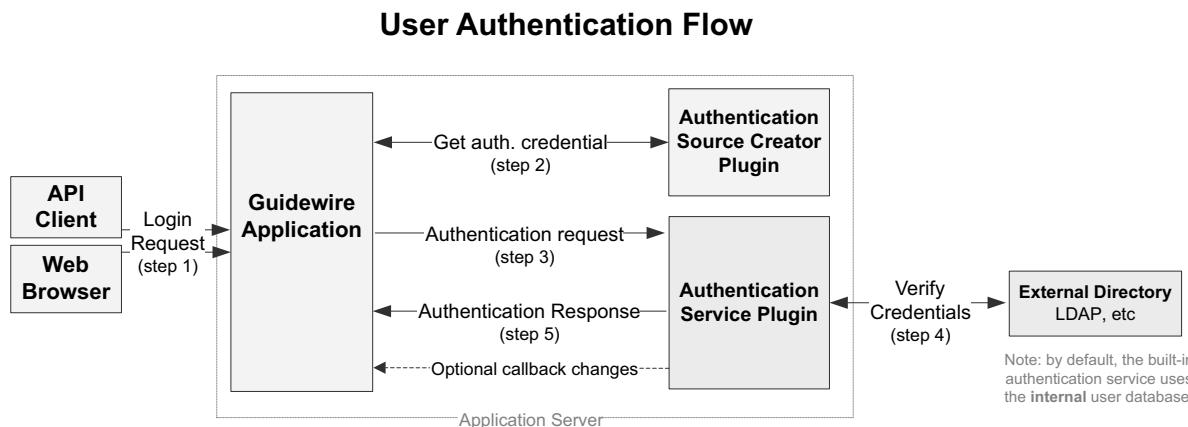
- Log in by using the web application user interface.
- Log in to the server through Guidewire Studio.
- Authenticate WS-I web service calls to the current server.

Authentication plugins must handle all types of logins other than WS-I web services. For authenticating WS-I web services, see the separate plugin interface in “Web Services Authentication Plugin” on page 59.

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:



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 to not need to throw exceptions.

If you do need to throw exceptions from your authentication source plugin implementation:

- 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.

- Only if that approach is insufficient for your login exception handling, you can create an entirely custom exception type as follows.
 - a. Create a subclass of `javax.security.auth.login.LoginException` for your authentication source processing exception.
 - b. Create a subclass of `gw.api.util.LoginForm`.
 - c. 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`.
 - d. 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, Such as the PolicyCenter PCF Login Page

In the default of implementation PolicyCenter, login-related PCF files set the username and password as HTTP request attributes. HTTP request attributes are hidden values in the request. Do not confuse HTTP attributes with URL parameters. To extract data from the URL itself, see “Authentication Data in Parameters in the URL” on page 184.

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");
```

This plugin interface provides only one method, which is called `createSourceFromHTTPRequest`. The following example of 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 from a URL with the syntax:

```
https://myserver:8080/pc/PolicyCenter.do?username=aapplegate&password=sheridan&objectID=12354
```

For example, use the following code:

```
package gw.plugin.security
uses javax.servlet.http.HttpServletRequest
uses com.guidewire.pl.plugin.security.AuthenticationSource
uses com.guidewire.pl.plugin.security.UserNamePasswordAuthenticationSource

class MyAuthenticationSourceCreatorPlugin implements AuthenticationSourceCreatorPlugin {

    override function createSourceFromHTTPRequest(request : HttpServletRequest) : AuthenticationSource {
        var source : AuthenticationSource
        var userName = request.getParameter( "username" )
        var password = request.getParameter( "password" )
        source = new UserNamePasswordAuthenticationSource( userName, password )

        return source
    }
}
```

Authentication Data in HTTP Headers for HTTP Basic Authentication

The `PolicyCenter/java-api/examples` directory also contains a simple example implementation of a plugin that gets authentication information from HTTP basic authentication. Basic authentication encodes the data in HTTP headers for some web servers, such as IBM’s WebSeal.

The example implementation turns this information into an Authentication Source if it is sent encoded in the HTTP request header, for example if using IBM’s WebSeal. The plugin decodes a username and password stored in the header and constructs a `UserNamePasswordAuthenticationSource`.

This plugin interface provides only one method, which is called `createSourceFromHTTPRequest`. The following example shows 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;  
        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");  
        }  
    }  
}
```

You can implement this authentication source creator interface and store more complex credentials. If you do this, you must also implement an authentication service that knows how to handle these new sources. To do that, implement a user authentication service plugin (`AuthenticationServicePlugin`), described in the next section.

To view the source code to this example, refer to

[PolicyCenter/java-api/examples/plugins/authenticationsourcecreator/](#)

User Authentication Service Plugin

An authentication service plugin (`AuthenticationServicePlugin`) implementation defines an external service that could authenticate a user. Typically, this would involve sending authentication credentials encapsulated in an *authentication source* and sending them to some separate centralized server on the network such as an LDAP server. This plugin is in the `gw.plugin.security` package namespace.

There are several mechanisms to log in to PolicyCenter:

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

Any `AuthenticationServicePlugin` implementation must handle all types of logins other than WS-I web services. For authenticating WS-I web services, see the separate plugin interface in “Web Services Authentication Plugin” on page 59.

For Guidewire Studio users and SOAP API calls, the credentials passed to this plugin are the standard `UserNamePasswordAuthenticationSource`, which contains a basic username and password. In addition, if you design a custom authentication source with data extracted from a web application login request, this plugin must be able to handle those credentials too.

There are three parts of implementing this 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 (`CallbackHandler`) in this method. 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` because this source is used by Guidewire Studio if connecting to the PolicyCenter server. A custom implementation must also support any other authentication sources that may be created by your custom *authentication source creator plugin*, if any.

Almost every authentication service plugin uses the `CallbackHandler` provided to it, if only to look up the public ID of the user after verifying credentials. Find the Javadoc for this interface in the class `AuthenticationServicePluginCallbackHandler`. This utility class includes four utility methods:

- `findUser` – Lets your code look up a user's public ID based on login user name
- `verifyInternalCredential` – Supports testing a username and password against the values stored in the main user database. This method is used by the default authentication service.
- `modifyUser` – After getting current user data and making changes, perhaps based on contact information stored externally, this method allows the plugin to update the user's information in PolicyCenter.

For more details of the method signatures, refer to the Java API Reference Javadoc for `AuthenticationServicePlugin`.

Authentication Service Sample Code

The product includes the following example authentication services in the directory `PolicyCenter/java-api/examples/plugins/authenticationservice`.

- **Default PolicyCenter authentication example** – This is the basic default service for PolicyCenter provided as source code. It checks the username and password against information stored in the PolicyCenter database and authenticates the user if a match is found.
- **LDAP authentication example** – `LDAPAuthenticationServicePlugin` is an example of authenticating against an LDAP directory.
- **JAAS authentication example** – `JAASAuthenticationServicePlugin` is an example of how you could write a plugin 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.

In the JAAS example, the plugin makes a `context.login()` call to the configured JAAS provider. The provider in turn calls back to the `JAASCallbackHandler` to request credential information needed to make a decision. The plugin provides this callback handler to the JAAS provider. This is a JAAS object, different from the callback handler provided by PolicyCenter to the plugin as a PolicyCenter API. If the user cannot authenticate to JAAS, then the JAAS provider throws an exception. Otherwise, login continues.

Error Handling in Authentication Service Plugins

Your code must be very careful to catch and categorize different types of errors that can happen during authentication. This is important for preserving this information for logging and diagnostic purposes. Additionally, PolicyCenter provides built-in features to clarify for the user what went wrong.

In general, the login user interface displays a default general message for failure to authenticate and ignores the text in the actual exception. However, `DisplayableLoginException` allows you to throw a more specific message to show 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. In general, PolicyCenter classes in the `com.guidewire` hierarchy are internal and not for customer use. However, the classes in the `com.guidewire.pl.plugin` hierarchy listed below are supported for customer use.

Exception name	Description
<code>javax.security.auth.login.FailedLoginException</code>	Throw this standard Java exception if the user entered an incorrect password.
<code>com.guidewire.pl.plugin.security.InactiveUserException</code>	This user is inactive.
<code>com.guidewire.pl.plugin.security.LockedCredentialException</code>	Credential information is inaccessible because it is locked for some reason. For example, if a system is configured to permanently lock a user out after too many attempts. Also see related exception <code>MustWaitToRetryException</code> .
<code>com.guidewire.pl.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. Also see related exception <code>LockedCredentialException</code> .
<code>com.guidewire.pl.plugin.security.AuthenticationException</code>	Other authentication issues not otherwise specified by other more specific authentication exceptions.
<code>com.guidewire.pl.plugin.security.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. For example, throw the exception as follows: <code>throw new DisplayableLoginException("The application shows this custom message to the user")</code>

You can make subclasses of the following exception classes:

- If `DisplayableLoginException` does not meet your needs, you can subclass the Java base class `javax.security.auth.login.LoginException`. To ensure the correct text displays for that class:
 - a. Create a subclass of `gw.api.util.LoginForm`.
 - b. 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`.
 - c. 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.
- You can also subclass the exception `guidewire.pl.plugin.security.DisplayableLoginException` if necessary for tracking unique types of authentication errors with custom messages.

SOAP API User Permissions and Special-Casing Users

Guidewire recommends creating separate a PolicyCenter user (or users) for SOAP API access. This user or set of users must have the minimum permissions allowable to perform SOAP API calls. Guidewire strongly recommends this user have few permissions or no permissions in the web application 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 sample shows 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, "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 (StringUtil.isNotBlank(_domainName)) {
        userName = _domainName + "\\" + userName;
    }

    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 Studio plugin editor.

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. See “*Messaging Editor*” on page 153 in the *Configuration Guide*. 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.

In the main config.xml file but not the plugin registry there is a sessiontimeoutsecs parameter that configures the duration of inactivity in seconds to allow before requiring reauthentication. This timeout period controls both the user’s web (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.

The following is an example of this inactive session timeout parameter:

```
<param name="SessionTimeoutSecs" value="10800"/>
```

Database Authentication Plugins

You might want the PolicyCenter server to connect to an Enterprise database, but require a flexible database authentication system. Or, you might be concerned about sending passwords as plaintext passwords openly across the network. Solve either of these problems by implementing a *database authentication plugin*.

A custom database authentication plugin can retrieve name and password information from an external system, encrypt passwords, read password files from the local file system, or any other desired action. The resulting username and password substitutes into the database configuration file anywhere that \${username} or \${password} are found in the database parameter elements.

It is important to understand that *database authentication plugins* are different from *user authentication plugins*. Whereas user authentication plugins authenticate users into PolicyCenter (from the user interface or using API), database authentication plugins help the PolicyCenter server connect to its database server.

To implement a database authentication plugin, implement a plugin that implements the class DBAuthenticationPlugin, which is defined in the Java package com.guidewire.p1.plugin.dbauth.

This class has only one method you need to implement: retrieveUsernameAndPassword, which must return a username and password. Store the username and password combined together as properties within a single instance of the class UsernamePasswordPair.

The one method parameter for retrieveUsernameAndPassword is the name of the database (as a String) for which the application requests authentication information. This will match 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 Studio configuration of your plugin. Get these parameters in your plugin implementation using the standard setParameters method of InitializablePlugin. For more information, see “Example Gosu Plugin” on page 129.

The username and password that this method returns need not be a plaintext username and password, and it typically would **not** be plaintext. A plugin like this typically encodes, encrypts, hashes, or otherwise converts the data into a secret format. The only requirement is that your database (or an intermediate proxy server that pretends to be your database) knows how to authenticate against this username and password.

The following example demonstrates this method by pulling this information from a file:

```
public class FileDBAuthPlugin implements DBAuthenticationPlugin, InitializablePlugin {
```

```

private static final String PASSWORD_FILE_PROPERTY = "passwordfile";
private static final String USERNAME_FILE_PROPERTY = "usernamefile";

private String _passwordfile;
private String _usernamefile;

public void setParameters(Map properties) {
    _passwordfile = (String) properties.get(PASSWORD_FILE_PROPERTY);
    _usernamefile = (String) properties.get(USERNAME_FILE_PROPERTY);
}

public UsernamePasswordPair retrieveUsernameAndPassword(String dbName) {
    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;
}
}

```

Guidewire PolicyCenter includes an example database authentication plugin that simply reads a username and password from files specified in the `usernamefile` or `passwordfile` parameters that you define in Studio.

PolicyCenter replace the `${username}` and `${password}` values in the `jdbcURL` parameter with values returned by your plugin implementation. For this example, the values to use are the text of the two files (one for username, one for password).

For the source code, refer to the `FileDBAuthPlugin` sample code in the `examples.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. See “[Messaging Editor](#)” on page 153 in the *Configuration Guide*. 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.

ContactManager Authentication

For more information about ContactManager authentication, see “[Configuring ContactManager Authentication with Core Applications](#)” on page 78 in the *Contact Management Guide*.

Document Management

PolicyCenter provides a user interface and integration APIs for creating documents, downloading documents, and producing automated form letters. You can integrate PolicyCenter with a separate external document management system (DMS) that stores the documents. Optionally, the external DMS can store the document metadata, such as the list of documents and their file types.

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. See “Policy Forms” on page 471 in the *Application Guide* and “Forms Integration” on page 423.

This topic includes:

- “Document Management Overview” on page 191
- “Choices for Storing Document Content and Metadata” on page 193
- “Document Storage Plugin Architecture” on page 195
- “Implementing a Document Content Source for External DMS” on page 196
- “Storing Document Metadata In an External DMS” on page 199
- “The Built-in Document Storage Plugins” on page 200
- “Asynchronous Document Storage” on page 201
- “APIs to Attach Documents to Business Objects” on page 202
- “Retrieval and Rendering of PDF or Other Input Stream Data” on page 203

See also

- For general information on plugins, which are an important part of document management in PolicyCenter, see “Plugin Overview” on page 123.
- “Document Production” on page 205

Document Management Overview

The PolicyCenter user interface provides a **Documents** section within the policy file.

The **Documents** section 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. 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. You can optionally integrate PolicyCenter with an external document management system (DMS).

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) may be slow. If so, synchronous actions with large documents may 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. For more information, refer to “Asynchronous Document Storage” on page 201.

It is important to understand the various types of IDs for a **Document** entity instance:

- **Document.PublicID** – the unique identifier for a single document in the DMS. If the DMS supports versions, the **PublicID** property does not change for each new version.
- **Document.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. See also the reference to extension properties later in this bullet list.
- **extension properties** – You can extend the base **Document** entity to contain version-related properties or other properties if it makes sense for your DMS.
- **Document.Id** – *This is an internal PolicyCenter property.* Never use the **Id** property to identify a document. Never get or set this property for document management.

Document Production Overview

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. Typically you would persist the document and attach the resulting new document to a business data object.

There are two ways to create documents:

- PolicyCenter can facilitate creation of documents on the user’s desktop using local application such as Microsoft Word or Excel. This is *client-side document production*.
- PolicyCenter can create some types of new documents from a server-stored template without user intervention. This is *server-side document production*.

For more information, see “Document Production” on page 205.

Document Retrieval Mode Overview

There are several document retrieval modes, also known as response types:

- **URL** – A URL to a local content store to display the content. The **URL** response type is the recommended response type if your DMS supports it. The **URL** response type permits the highest performance for PolicyCenter.

- DOCUMENT_CONTENTS – An input stream that contains the raw document contents. You can enable web viewing of arbitrary input streams such as PDF data. See “Retrieval and Rendering of PDF or Other Input Stream Data” on page 203

For more information, see “Retrieving Documents” on page 197.

Choices for Storing Document Content and Metadata

There are two separate kinds of data that a document storage system (DMS) processes for PolicyCenter. Before implementing a document management integration, you must decide where to store document content and metadata. The following table compares and contrasts these kinds of data.

Type of data	Description	Plugin interface that handles this data	Default behaviors
Document content	<p>Document content can be anything that represents incoming or outgoing documents.</p> <p>For example:</p> <ul style="list-style-type: none"> • a fax image • a Microsoft Word file • a photograph • a PDF file <p>A <i>document content source</i> is code that stores and retrieves the document content.</p>	IDocumentContentSource	<p>It is important to note that for the highest performance and data integrity, use an external DMS.</p> <p>In the initial PolicyCenter configuration, built-in classes implement asynchronous storage onto the local file system. For more information, see “The Built-in Document Storage Plugins” on page 200 and “Asynchronous Document Storage” on page 201. If you use an external DMS, you would not use local file system storage.</p>
Document metadata	<p>Document metadata describes the file:</p> <ul style="list-style-type: none"> • name, which typically is the file name • MIME type • description • the full set of metadata on the DMS so users can search for documents. • which business objects are associated with each document • other fields defined by the application or customer extensions, such as recipient, status, or security type <p>A <i>document metadata source</i> is code that manages and searches document metadata.</p>	IDocumentMetadataSource	<p>In the default PolicyCenter configuration, this plugin is disabled. In other words, in the Studio user interface for this plugin interface, the Enabled checkbox is unchecked.</p> <p>If this plugin is disabled, PolicyCenter stores the document metadata internally in the database, with one Document entity instance for each document. If you store the metadata internally in this way, searching for documents is very fast. Many DMS systems are not designed to withstand high capacity for external searching and metadata lookup. Additionally, if you use internal metadata, the metadata is available even if the DMS is offline.</p> <p>There is a built-in plugin implementation called LocalDocumentMetadataSource. This is for testing only and is unsupported in production. To correctly configure fast internal document metadata, uncheck the Enabled checkbox for this plugin.</p>

Deciding Where to Store Document Content and Metadata

The best approach for configuring these plugins depends on whether you want to use an external DMS and how your company uses documents. For the highest performance and data integrity, use an external DMS.

To configure PolicyCenter for internal document storage (no external DMS):

1. For document content, use the default implementations of the `IDocumentContentSource` plugin. For more information, see “The Built-in Document Storage Plugins” on page 200 and “Asynchronous Document Storage” on page 201.
2. For document metadata, do not enable the `IDocumentMetadataSource` plugin. If this plugin is disabled, PolicyCenter stores the document metadata internally in the database, with one `Document` entity instance for each document. Internal metadata storage makes document search fast. Additionally, if you use internal metadata, you can search document metadata even when the DMS is offline.

WARNING Choose your document metadata location carefully. See “Internal Versus External Metadata Permanently Affects Some Objects” on page 194.

To configure PolicyCenter for an external DMS:

1. For the document content, write your own implementation of the `IDocumentContentSource` plugin to store and retrieve content. Your plugin implementation stores and retrieves files using the proprietary API for your external DMS. If you use an external DMS, you can still use the built-in asynchronous document storage system. See “Asynchronous Document Storage” on page 201.
2. For the document metadata, the best solution depends on how you use documents:

WARNING Choose your document metadata location carefully. See “Internal Versus External Metadata Permanently Affects Some Objects” on page 194.

- If most documents in the DMS related to PolicyCenter exist because users upload or create documents in PolicyCenter, use internal metadata storage. Do not enable the `IDocumentMetadataSource` plugin in Studio. If this plugin is disabled, PolicyCenter stores the document metadata internally in the database, with one `Document` entity instance for each document. Internal metadata storage using the database makes document search very fast. Additionally, if you use internal metadata, the metadata is available when the DMS is offline.
- If most documents in the DMS related to PolicyCenter are added directly into the DMS without PolicyCenter, write your own implementation of the `IDocumentMetadataSource` plugin. Your plugin implementation manages and searches the metadata. If you want to share document metadata searching across multiple Guidewire applications, it is best to use external metadata.

Internal Versus External Metadata Permanently Affects Some Objects

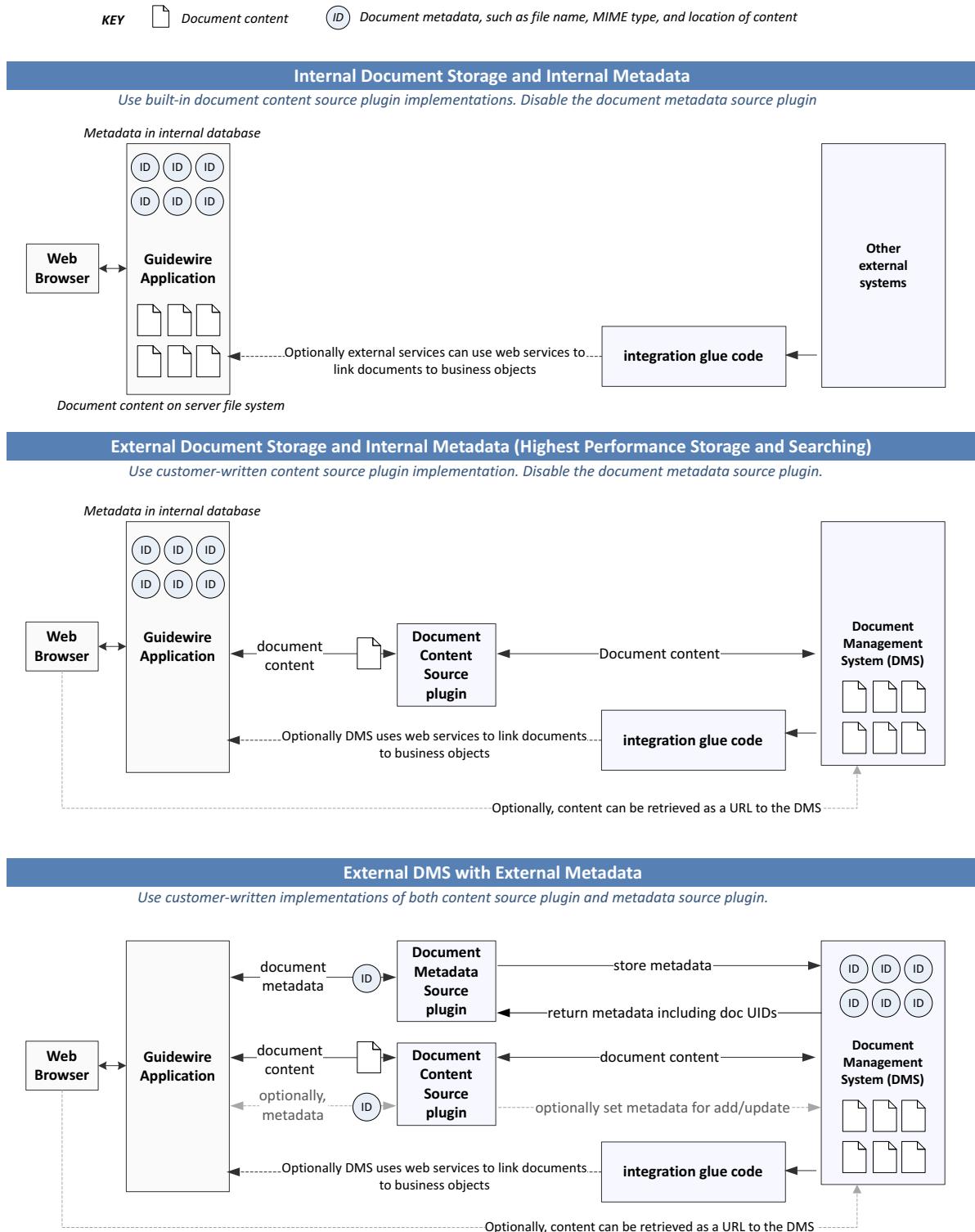
Whichever approach to handling document metadata you choose (internal or external), be aware that you must continue that approach permanently. Note objects and other similar text blocks reference a document within text as a specially-formatted hyperlink within the text itself. The format of this hyperlink within text blocks is slightly different based on whether PolicyCenter or a remote system handles the document metadata. If this quality changes from internal to external, or from external to internal, those document hyperlinks fail.

If you are considering this type of transition, contact Guidewire Customer Support for advice before proceeding.

Document Storage Plugin Architecture

The following diagram summarizes implementation architecture options for document management.

Document Storage Architecture Options



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*. For the list of options, see “Document Retrieval Mode Overview” on page 192.

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.

Adding Documents and Metadata

A new document content source plugin must fulfill a request to add a document by implementing the `addDocument` method. This method adds a new document object to the repository.

IMPORTANT If you enable and implement the `IDocumentDataSource` plugin, PolicyCenter also calls this method to update an existing document. Write your code to accommodate being called with an existing document.

The method takes as arguments:

- the document metadata in a `Document` object
- the document contents as an `InputStream` object

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 plugin must 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 must set the modified date in the `Document.DateModified` property.

The method must persist these changes to the PolicyCenter database.

The return value from the `addDocument` method is very important:

- If you do not enable and implement the `IDocumentDataSource` plugin, you must return `false`.
- If you enable and implement the `IDocumentDataSource` plugin implementation:
 - If you stored metadata from the `Document` object in addition to document content, return `true`.
 - Otherwise, return `false`. PolicyCenter calls the registered `IDocumentDataSource` plugin implementation to store the metadata.

For related discussion, see “Deciding Where to Store Document Content and Metadata” on page 193.

Error Handling

If your plugin code encounters errors during storage before returning from this method, throw an exception from this method. If document storage errors are detected later, such as for a asynchronous document storage, the document content storage plugin must handle errors in an appropriate fashion. For example, send administrative e-mails or create new activities to investigate the problem. You could optionally persist the error information and design a system to track document creation errors and document management problems in a separate user interface for administrators.

Retrieving Documents

A new document content source plugin must fulfill a request to retrieve a document. To fulfill this kind of request, implement the `getDocumentContentsInfo` method. This method takes a `Document` object as a parameter. Use the `Document.DocUID` property to identify the target document within the external repository. Your code could use additional data model extensions on `Document` as needed.

The `getDocumentContentsInfo` method can read the `Document` object but must not modify the `Document` object.

The plugin must return its results within an instance of `gw.document.DocumentContentsInfo`, which is a simple object with the following properties:

- **Response type** – An enumeration in the `ResponseType` property indicates the type of the data:
 - URL – A URL to a local content store to display the content. For non-editable documents, the URL response type is the recommended response type if your DMS supports it. The URL response type permits the highest performance for PolicyCenter. However, the URL response type is problematic for editing documents, and would require modification of PCF code to accommodate it.
 - DOCUMENT_CONTENTS – An input stream that contains the raw document contents. You can enable web viewing of arbitrary input streams such as PDF data. See “Retrieval and Rendering of PDF or Other Input Stream Data” on page 203
- **Hidden target frame as Boolean** – For response types other than DOCUMENT_CONTENTS, the Boolean property `TargetHiddenFrame` specifies whether PolicyCenter opens the web page in a hidden frame within the user’s web browser. For example, if the document repository is naturally implemented as a small web page that contains JavaScript, set this property to `true`. JavaScript code from that page might open another web page as a popup window that displays the real document contents.
- **Contents as input stream** – The `InputStream` property contains the document contents in the form of an input stream, which is raw bytes as an `InputStream` object.

The `includeContents` Boolean Parameter

The `getDocumentContentsInfo` method has a Boolean parameter `includeContents`.

- If this parameter is `true`, include an input stream with the contents of the document. Set only the properties, `ResponseType`, `TargetHiddenFrame`, `InputStream`.
- If this parameter is `false`, only the response type is needed. Set only the property `ResponseType`. Do not set any other properties.

PolicyCenter calls some methods in a `IDocumentContentSource` plugin twice for each document: `isDocument` and `getDocumentContentsInfo`. Your plugin implementation must support this design. For the `getDocumentContentsInfo` method, the `includeContents` parameter is set to `false` on the first call and `true` on the second call. Your plugin must return a `DocumentContentsInfo` object with the same response type in response to both calls. Returning any other result is unsupported and results in undefined behavior.

MIME Type

The `DocumentContentsInfo.ResponseMimeType` property includes a MIME type. However, your `getDocumentContentsInfo` method must ignore this property. PolicyCenter sets the MIME type based on the `Document` properties after calling the `getDocumentContentsInfo` method. However, it is unsupported to rely on its value at the time that PolicyCenter calls `getDocumentContentsInfo`.

Checking for Document Existence

A document content source plugin must fulfill requests to check for the exist 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 twice for each document: `isDocument` and `getDocumentContentsInfo`. Your plugin implementation must support this design.

Removing 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 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 successful 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 `false`.

Updating 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 changed 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 appropriate.

IMPORTANT During document update, you can optionally set `Document.DocUID` and any extension properties that represent versions. However, do not change the `Document.PublicID` property. On a related note, never get or set the `Document.ID` property. See “Document Storage Overview” on page 192 for related discussion.

The return value from the `updateDocument` 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:
 - 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.

For related discussion, see “Deciding Where to Store Document Content and Metadata” on page 193.

Storing Document Metadata In an External DMS

In the default PolicyCenter configuration, the `IDocumentMetadataSource` plugin is disabled. In other words, in the Studio user interface for this plugin interface, the `t` checkbox is unchecked. If this plugin is disabled, PolicyCenter stores the document metadata locally in the database, with one `Document` entity instance for each document. If you store the metadata locally in this way, searching for documents is very fast. Many DMS systems are not designed for high capacity for external searching and metadata lookup. Additionally, if you use local metadata, the metadata is available even if the DMS is offline.

Optionally you can store document metadata in the document source content plugin so the content plugin handles the content and the metadata.

IMPORTANT It is critical to carefully read “Choices for Storing Document Content and Metadata” on page 193 before beginning implementation.

If you are sure you want to store document metadata in the DMS, write a class that implements the document metadata source (`IDocumentMetadataSource`) plugin interface. Typically this plugin sends the metadata to the same external system as the document content source plugin, but it does not have to do so.

It is important to understand the various of IDs for a `Document` entity instance:

- `Document.PublicID` – the unique identifier for a single document in the DMS. If the DMS supports versions, the `PublicID` property does not change for each new version.
- `Document.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. See also the reference to extension properties later in this bullet list.
- **extension properties** – You can extend the base `Document` entity to contain version-related properties or other properties if it makes sense for your DMS.
- `Document.Id` – *This is an internal PolicyCenter property.* Your implementation of the document metadata source plugin must never track documents by the ID (`Document.Id`) property.

IMPORTANT Always track a document by using the `PublicID` property, not the `Id` property.

The required plugin methods include:

- `saveDocument` – Persist document metadata in a `Document` object to the document repository. 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.
- `retrieveDocument` – Returns a completely initialized `Document` object with the latest information from the repository.
- `searchDocuments` – Return the set of documents in the repository that match the given set of criteria. This method’s parameters include a `RemotableSearchResultSpec` entity instance. This object contains sorting and paging information for the PCF list view infrastructure to specify results.

IMPORTANT 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.

- `removeDocument` – Remove a document metadata in a `Document` object from the document repository.

See also

- For more details, refer to the API Reference Javadoc documentation for `IDocumentMetadataSource`.
- For more information about the document source content plugin, see “Implementing a Document Content Source for External DMS” on page 196.
- For more information about the included reference implementation, see “The Built-in Document Storage Plugins” on page 200.

The Built-in Document Storage Plugins

PolicyCenter includes a reference implementation for the `IDocumentContentSource` plugin. This plugin implementation simply stores documents on the PolicyCenter server’s file system. By default, PolicyCenter stores the document metadata such as document names in the database. Document names are what people typically think of as document file names. You can override this behavior by implementing the `IDocumentMetadataSource` plugin.

To decide whether to use the built-in document storage plugin for your project, carefully read these topics:

- “Choices for Storing Document Content and Metadata” on page 193
- “Document Storage Plugin Architecture” on page 195

Built-in Document Storage Directory and File Name Patterns

The document content and storage plugins use the `documents.path` parameter in their XML definition files to determine the storage location. If the value of that parameter is an absolute path, then the specific location is used. If the value is a relative path, then the location is determined by using the value of the temporary directory parameter (`javax.servlet.context.tempdir`) is used instead.

The temporary directory property is the root directory of the servlet container’s temp directory structure. In other words, this is a directory that servlets can write to as scratch space but with no guarantee that files persist from one session to another. The temporary directory of the built-in plugins in general are for testing only. Do not rely on temporary directory data in a production (final) PolicyCenter system, and even the built-in plugins are for demonstration only, as indicated earlier.

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	<code>Account + accountPublicID</code>	<code>/documents/AccountABC:02154/</code>
Policy	<code>accountDirectory + "/" + Policy + policyPublicID</code>	<code>/documents/AccountABC:02154/PolicyABC:02154/</code>
PolicyPeriod	<code>policyDirectory + "/" + PolicyPeriod + policyPeriodPublicID</code>	<code>/documents/AccountABC:02154/PolicyABC:02154/PolicyPeriodABC:02154/</code>

Notice that for all entities, the public ID is used. No other types of IDs 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 and it 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 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 built-in document metadata source plugin provides a unique document ID back to PolicyCenter. That document ID identifies the file name and relative path within the repository to the document.

For example, a policy document's repository-relative path might look something like this:

/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. A separate thread on the batch server sends documents to your real document management system using the messaging system.

To implement this, PolicyCenter includes two software components:

- **Asynchronous document content source** – PolicyCenter includes a document content source plugin implementation `gw.plugin.document.impl.AsyncDocumentContentSource`. When it 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.
- **Document storage messaging transport** – A separate part of the system uses PolicyCenter messaging to send documents to the external system. In a clustered environment, any server can create (submit) a new message. However, only the batch server runs the messaging plugins to actually send messages across the network. In the built-in configuration, the transport calls the document storage plugin implementation that you implement. As a result, in a clustered environment, your document content source implementation only runs on the batch server.

By default, the asynchronous document storage plugin is enabled. By default, it simply uses the built-in document content storage implementation. To write your own document content source plugin and configure it for use with asynchronous sending, see “Configuring Asynchronous Document Storage” on page 202.

For maximum document data integrity and document management features, use a complete commercial document management system (DMS) and implement new `IDocumentMetadataSource` and `IDocumentContentSource` plugins to communicate with the DMS. Where at all possible, store the metadata in the DMS if it can natively store this information. In all cases, store the metadata in only one location: either in the DMS or PolicyCenter built-in metadata storage, but not both. Avoid duplicating metadata in the PolicyCenter database itself for production systems.

Configuring Asynchronous Document Storage

To configure asynchronous sending with your own content source plugin

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 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.

To temporarily disable asynchronous sending

1. In the Studio Project window, navigate to `Configuration → config → Plugins → registry`, and then open `IDocumentContentSource.gwp`.
2. In the `Parameters` table, find the `TrySynchedAddFirst` parameter.
3. Set this parameter to `false`.

To permanently disable asynchronous sending

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. If your class was not implemented in Gosu, click the red minus sign to remove the Gosu plugin configuration. Click the green plus sign and choose either Java or OSGi.
3. Disable the messaging destination:
 - 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 `Enabled` checkbox.

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 types of objects (including a policy) using document-related Gosu APIs.

To create a new object in Java, use the `EntityFactory` class discussed in “Accessing Entity and Typecode Data in Java” on page 633.

You can create a new document with Java code such as:

```
Document doc = (Document) EntityFactory.getInstance().newEntity(Document.class);
```

To implement document search, the `searchDocument` method might look similar to the following example. This example creates a new `DocumentSearchResult` for the result set and a new `Document` for a specific document summary:

```
DocumentSearchResult result =
(DocumentSearchResult)EntityFactory.getInstance().newEntity(DocumentSearchResult.class);

Document document = (Document)EntityFactory.getInstance().newEntity(Document.class);
document.setName("Test Document");
document.setType(DocumentType.DIAGRAM);

...
result.addToSummaries(document);
```

Web Service APIs to Attach Documents to Business Objects

PolicyCenter provides a web services API for adding 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 e-mail 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 methods to add documents a policy revision:

- `addDocumentToPolicyPeriod` – Add a new document to a policy revision

Retrieval and Rendering of PDF or Other Input Stream Data

You can display arbitrary `InputStream` content in a window. For example, you can display a PDF returned from code that returns PDF data as a byte stream (`byte[]`) from a plugin, encapsulated in a `DocumentContentsInfo` object.

Use the following utility method from Gosu including PCF files:

```
gw.api.document.DocumentsUtil.renderDocumentContentsDirectly(fileName, docInfo)
```

There are other utility methods on the `DocumentsUtil` object that may be useful to you. Refer to the API reference documentation in the Studio Help menu for more details.

For more information about PDF creation, see “Server-side PDF Licensing” on page 212.

Document Production

PolicyCenter provides a user interface and integration APIs for creating documents, downloading documents, and producing automated form letters. The APIs include the ability to integrate the PolicyCenter document user interface with a separate corporate document management system that can store the documents and optionally the document metadata.

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. See “Policy Forms” on page 471 in the *Application Guide* and “Forms Integration” on page 423.

This topic includes:

- “Document Production Overview” on page 205
- “Document Template Descriptors” on page 213
- “Generating Documents from Gosu” on page 224
- “Template Web Service APIs” on page 226

See also

- For general information on plugins, which are an important part of document management in PolicyCenter, see “Plugin Overview” on page 123.
- For an overview of document management and storage, see “Document Management” on page 191

Document Production Overview

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.

Users can create new documents locally from a template and then attach them to a policy or other PolicyCenter objects. 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.

The `IDocumentProduction` interface is the plugin interface used to create new document contents from within the Guidewire application. It interfaces with a document production system in a couple of different ways, depending on whether the new content can be returned immediately or requires delayed processing. These two modes are:

- **Synchronous production** – The document contents are returned immediately from the creation methods.
- **Asynchronous production** – The creation method returns immediately, but the actual creation of the document is performed elsewhere and the document may not exist for some time.

There are different integration requirements for these two types of document production.

There are several document response types:

- **URL** – A URL to a local content store to display the content. The `URL` response type is the recommended response type if your DMS supports it. The `URL` response type permits the highest performance for PolicyCenter.
- **DOCUMENT_CONTENTS** – An input stream that contains the raw document contents. You can enable web viewing of arbitrary input streams such as PDF data. See “Retrieval and Rendering of PDF or Other Input Stream Data” on page 203

These settings are defined as the `ResponseType` property within `DocumentContentsInfo`, which is the return result from synchronous production methods.

The details of these plugins are discussed in “User Interface Flow for Document Production” on page 206. For locations of templates and descriptors, see “Template Source Reference Implementation” on page 223.

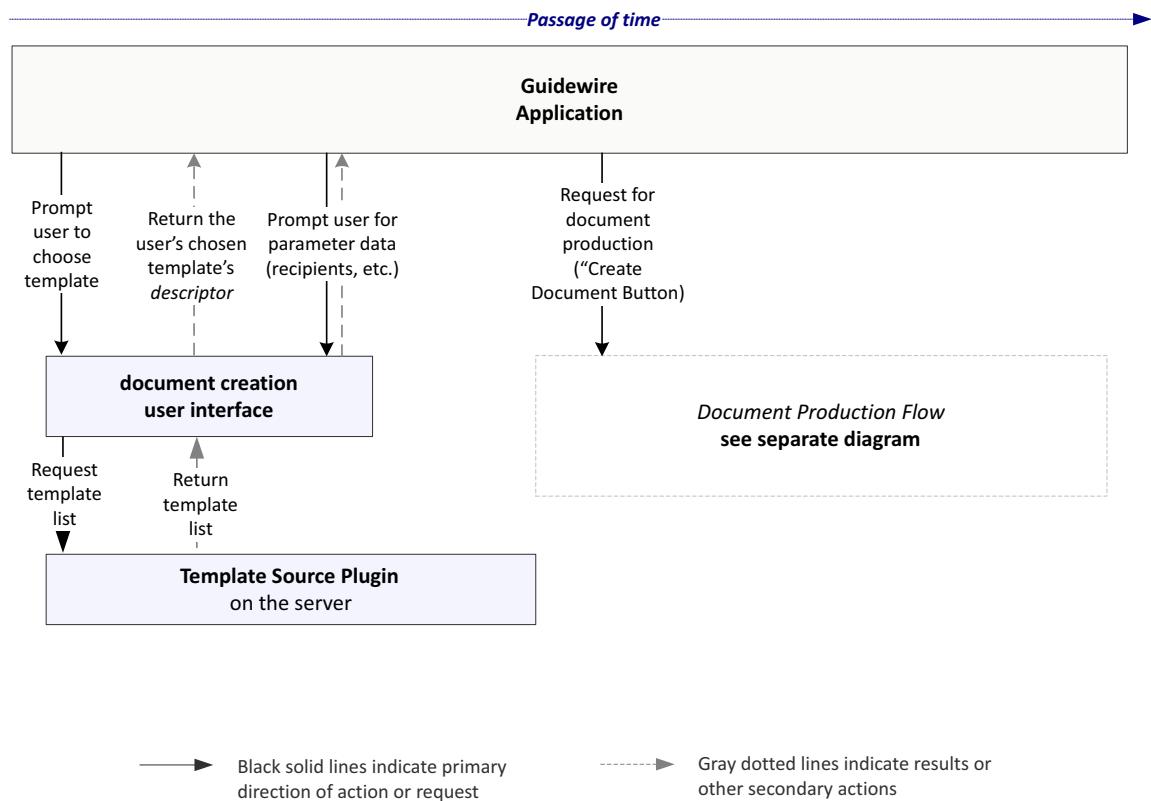
After document production, storage plugins store the document in the document management system.

User Interface Flow for Document Production

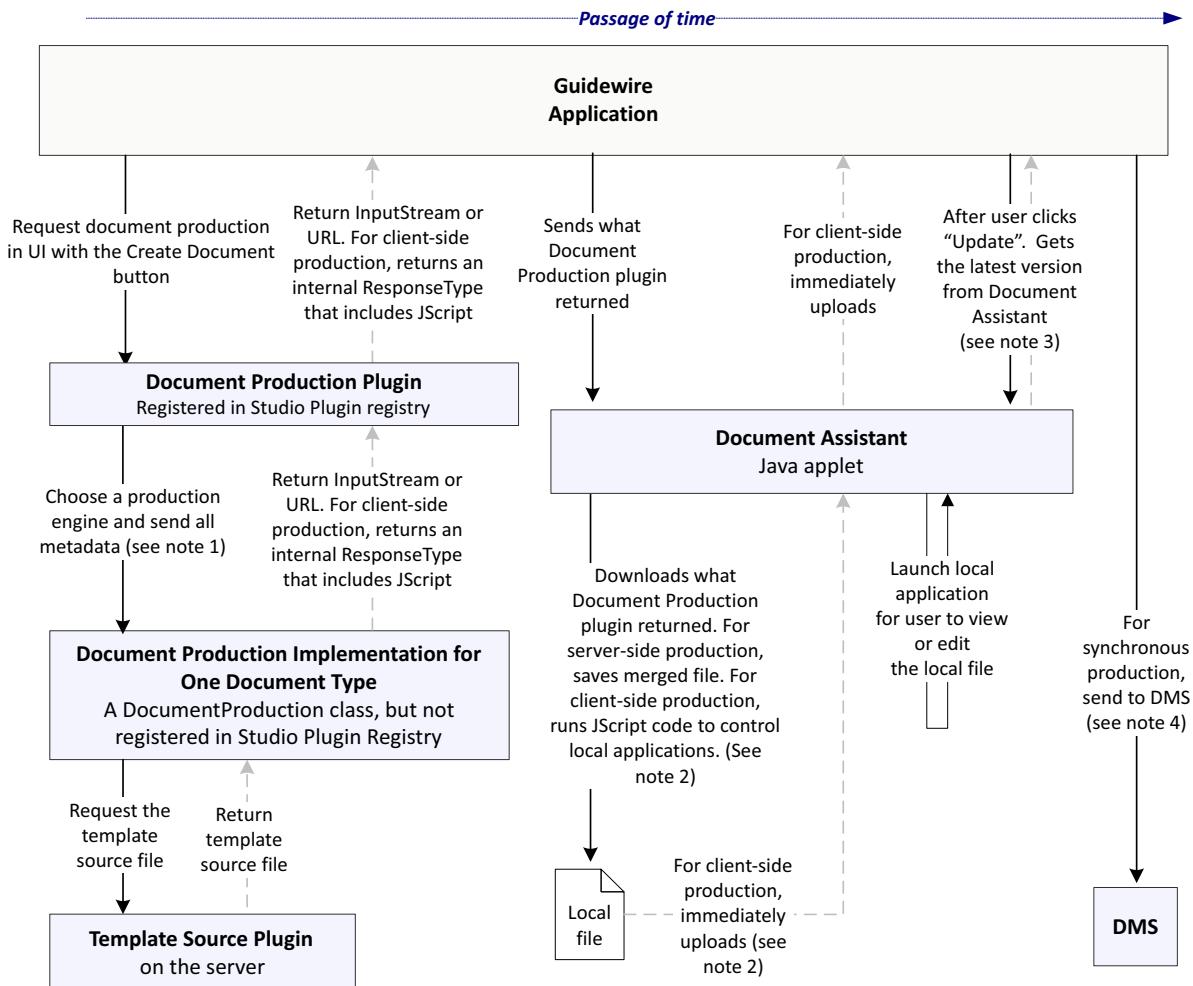
From the PolicyCenter user interface, create forms and letters and choose the desired template. You can select other parameters (some optional) to pass to the document production engine. PolicyCenter supports Gosu-initiated automatic document creation from business rules. The automatic and manual processes 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 illustrates the interactions between PolicyCenter, the various plugin implementations, the Document Assistant, and an external document management system to manage the generation of a form or letter.

Interactive Document Creation UI Flow



Document Production Flow



Notes

1. The template descriptor for this request determines the document production implementation. It specifies a template handler String or a MIME type, both of which map to a document production implementation.
2. For client-side production, if Document Assistant is not present, the Guidewire application sends the web browser a JScript file. The JScript file does the equivalent of Document Assistant if you properly configure the PC and approve the request. To upload, the user interface prompts you to locate the merged file on the local file system.
3. If the Document Assistant is not present, you must manually locate the merged file on the local file system to upload.
4. For asynchronous production, after completion, the document production plugin sends the document to the DMS.



Primary direction of action or request



Results or other secondary actions

The chronological steps are as follows:

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.
3. The template source returns a list of templates.
4. The document creation user interface displays a list of templates and lets you choose one, perhaps after searching or filtering among the list of templates. Searching and filtering uses the template metadata, which is information about the templates themselves.
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 parameters 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 parameter data to PolicyCenter.
7. PolicyCenter chooses the appropriate document production plugin from values in the template descriptor. The descriptor file can indicate a specific document production plugin or let one be chosen based on the MIME type of the file. The document production plugin gets a template descriptor and the parameter data values.
8. The document production plugin obtains the actual template file from the template source.
9. The document production plugin takes whatever steps are necessary to launch an editor application or other production engine for merging the template file with the parameter data. PolicyCenter includes document production plugins that process Microsoft Word files, Adobe Acrobat (PDF) files, Microsoft Excel files, Gosu templates, and plain text files. For Gosu templates, the Gosu document production plugin executes the Gosu template using the parameter data.
The result of this step is a *merged document*. It is possible to produce documents using applications on the user's computer, a process known as *client-side document production*. For example, the built-in production plugins use client-side production for Microsoft Word and Microsoft Excel documents.
10. If the document was created from the user interface, PolicyCenter automatically downloads the file to your local machine if the configuration parameter `AllowDocumentAssistant` is enabled. The file opens in the appropriate application on the desktop. You can print the file, and if the editor application supports editing, you can change the file and save it in the editor application. This describes the reference implementation, but the user interface flow can work differently if you customize the PCF files for a different workflow.
11. If the file is on your PC, you upload the completed, locally merged document, or you can choose an alternative document to upload. You can then specify some additional parameters about the document. If the configuration parameter `AllowDocumentAssistant` is enabled, the upload step by the Guidewire Document Assistant supplies the location of the locally merged document. If `AllowDocumentAssistant` is disabled or the Guidewire Document Assistant control is not installed, then you must browse manually to the local location of the merged document to upload.
12. Additionally, depending on the type of document production, the production engine could add the document to the document storage system. The production system or the storage system could notify PolicyCenter using plugin code or from an external system using the web services API. For more information about document storage, see "Document Management Overview" on page 191.

The preceding steps describe the reference implementation, but you can configure many parts of the flow to suite your needs. You can configure the user interface flow to work differently by modifying the user interface PCF configuration files. In some cases, you can configure the flow by modifying the document-related plugins.

Document Production Plugins

As mentioned in “User Interface Flow for Document Production” on page 206, PolicyCenter supports two types of document production:

- *synchronous production* – the document contents return immediately
- *asynchronous production* – the creation method returns immediately but creation happens later

These two types of document production have 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 is an interface to a document creation system for a certain type of document. The document creation process may involve extended workflow and/or asynchronous processes, and it may depend on `Document` entity or set properties in the `Document`.

There are some additional related interfaces that assist the `IDocumentProduction` plugin:

- `IDocumentTemplateSource` and `IDocumentTemplateDescriptor` – Encapsulate the basic interface for searching and retrieving templates that describe the document to be created.
The descriptive information includes the basic metadata (name, MIME type, and so on) and a pointer to the template content. Specifically, `IDocumentTemplateDescriptor` describes the templates used to create documents and `IDocumentTemplateSource` plugin actually lists and retrieves the document templates
- `IPDFMergeHandler` – Used in creation of PDF documents. View the built-in implementation to set parameters in the default implementation.

The `IDocumentProduction` plugin has two main methods: `createDocumentSynchronously` and `createDocumentAsynchronously`.

- The `createDocumentSynchronously` method returns document contents as:
 - URL – A URL to a local content store to display the content. The URL response type is the recommended response type if your DMS supports it. The URL response type permits the highest performance for PolicyCenter.
 - DOCUMENT_CONTENTS – An input stream that contains the raw document contents. You can enable web viewing of arbitrary input streams such as PDF data. See “Retrieval and Rendering of PDF or Other Input Stream Data” on page 203

These settings are defined as the `ResponseType` property within `DocumentContentsInfo`, which is the return result from synchronous production methods.

For documents created synchronously, the caller of the `createDocumentSynchronously` method must pass the contents to the document content plugin for persistence. The `Document` parameter can be modified if the `DocumentProduction` plugin wants to set any properties for persistence in the database.

- The `createDocumentAsynchronously` method returns the document status, such as a status URL that could display the status of the document creation. In the reference implementation, none of the built-in document production plugins support asynchronous document creation. The default user interface does not actually use the results of the `createDocumentAsynchronously` method. To support this type of status update, customize the user interface PCF files to generate a new user interface. Also, customize the included `DocumentProduction` Gosu class that generates documents from Gosu. By default, `DocumentProduction` does not use the result of the `createDocumentAsynchronously` method.
- 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. See “Web Service APIs to Attach Documents to Business Objects” on page 203.

If your code to add the document is running within the server in local Gosu or Java code, do not call the SOAP APIs to call back the same server. Calling back to the same server using SOAP is not generally safe. Instead, use domain methods on the entity to add the entity. For example:

```
newDocumentEntity = Policy.addDocument()
```

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

It is possible to produce documents using applications on the user's computer, a process known as *client-side document production*. For example, the built-in production plugins use client-side production for Microsoft Word and Microsoft Excel documents. The Document Assistant can call the local application to open the file and merge in necessary properties. On Windows, the Document Assistant sends JScript that can control local applications.

For more information about the built-in production plugins, see "Built-in Document Production Plugins" on page 224.

For more details on specific methods of these plugins, refer to the Java API Reference Javadoc. In the Javadoc, there are two versions of the `IDocumentProduction` interface. There is a base class Guidewire platform version with ".pl." in the fully-qualified package name. To implement the plugin, you must implement the other version, the PolicyCenter-specific version, which has ".pc." in its fully-qualified package name.

Configuring Document Production and MIME Types

Document production configuration is defined by the registry for the `IDocumentProduction` plugin in the Plugins editor in Studio. The registry includes a list of parameters that define *template types* and their corresponding `IDocumentProduction` plugin implementations. In the registry, a template type is either a MIME type that PolicyCenter recognizes or a text value that you provide to define a template type.

Defining a Template Type for Document Production

You define template types by adding parameters to the list in the registry for the `IDocumentProduction` plugin. Each parameter in the list specifies the name of a template type and a class that implements the `IDocumentProduction` plugin interface to handle production for that template type. Separately, you must deploy the implementation class to the `PolicyCenter/modules/configuration/plugins/document/classes` directory and necessary libraries to the `PolicyCenter/modules/configuration/plugins/document/lib` directory.

Built-in Implementations of the Document Production Plugin

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

`com.guidewire.pc.plugin.document.internal.WordDocumentProductionImpl` as the implementation class.

Refer to the plugin registry for the `IDocumentProduction` plugin to see additional built-in plugin implementations.

How PolicyCenter Determines which Plugin Implementation to Use for Document Production

PolicyCenter determines which `IDocumentProduction` implementation to use to produce a document from a specific template by following this procedure:

1. PolicyCenter searches for a non-MIME-type entry in the plugin parameter list that matches the `documentProductionType` property of the template. If a match is found, PolicyCenter proceeds with document production.
2. If no match is found, PolicyCenter searches for a MIME-type entry in the plugin parameter list that matches the `mimeType` property of the template. If a match is found, PolicyCenter proceeds with document production.
3. If no match is found, document production fails.

You can replace the default dispatching implementation with your own `IDocumentProduction` implementation.

Server-Side PDF Document Production

PolicyCenter supports server-side PDF production but not client-side PDF production.

Adding a custom MIME type for Document Production

Adding a custom MIME type that PolicyCenter recognizes requires several steps.

1. In `config.xml`, add the new MIME type to the `<mimetypemapping>` section. The information for each `<mimetype>` element contains these 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.

How you add a MIME type depends on the brand of application server. For Tomcat, you configure MIME types in the `web.xml` configuration file by using `<mime-mapping>` elements. Assure the MIME type that you need is not in the list already you try to add it.

See also

- For basic instructions, see “Using the Plugins Registry Editor” on page 131 in the *Configuration Guide*.
- For more about information built-in `IDocumentProduction` implementations classes, see “Built-in Document Production Plugins” on page 224.

Server-side PDF Licensing

In the PolicyCenter reference implementation, the server-side PDF production software is made by a company called Big Faceless Org (BFO). Without a license, the generated documents contain a large watermark that reads “DEMO” on the face of each generated page, rather than preventing document creation entirely. To remove the watermark, you must obtain a license key through Guidewire Customer Support.

All server-side PDF production licenses are licensed per server *CPU*, not per server.

With your copy of PolicyCenter, you are entitled to four CPU licenses for PDF production. However, you must still go through a process to obtain the keys associated with your licenses. In addition, you may purchase additional licenses through the same process.

To obtain a license key:

1. Contact your Customer Support Partner or email support@guidewire.com with your request for a PDF production license for PolicyCenter.
2. A support engineer sends you an Authorization Form.
3. Fill out the Authorization Form, including the number of CPUs to use. For multi-CPU servers, include the total number of CPUs.
4. Fax the filled-out form to Guidewire prior to issuing the license.
5. The Guidewire support engineer requests license keys from the appropriate departments.
6. 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.

Configuration

The default configuration appears in the Plugins registry in Studio. In the Project window, navigate to Configuration → config → Plugins → registry, and then open IPDFMergeHandler.gwp. For information about using the plugins editor, see “Using the Plugins Registry Editor” on page 131 in the *Configuration Guide*.

Ensure the implementation class is set to `gw.plugin.document.impl.BFOMergeHandler`, which is the default.

In the list of plugin parameters, set the following two plugin parameter values:

- Set plugin parameter `BatchServerOnly` value to `true` for typical usage. The `BatchServerOnly` parameter determines whether PDF production happens on each server or solely on the batch server.
- Set 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. In a clustered environment you must also set the license key value in the `PDFMergeHandlerLicenseKey` parameter in `config.xml`.

PolicyCenter uses the `IPDFMergeHandler` plugin interface only for server-side PDF production in the default implementation of the `IDocumentProduction` interface.

Document Template Descriptors

A *document template descriptor* describes an actual document template, such as a Microsoft Word mail merge template, an Adobe PDF form, or a Gosu template. An interface called `IDocumentTemplateDescriptor` defines the API for an object that represents a document template descriptor.

IMPORTANT In most cases, it is best to use the built-in implementation of the `IDocumentTemplateDescriptor` interface. The built-in implementation reads template information from a standard XML file. You can modify XML data if desired. However, most customers do not need to modify the code that reads or writes the XML file. Be aware that 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 four different kinds of information:

- **Template metadata** – Template metadata is metadata about the template itself, not the file. Template metadata includes the template ID, the template name, and the calendar dates that limit the availability of the template.
- **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** – Context objects are objects that can be inserted into the document template, or have properties extracted from them before inserting them into the document template. For example, an email document template might include To and CC recipients as context objects, each of which is of the type `Contact`. The context objects include default values to be inserted, as well as a set of legal alternative values for use in the PolicyCenter document creation user interface. A context object is an object attached to the merge request that can either be inserted into the document or certain properties within the object extracted from it.
- **Property names and values to merge** – 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`. The template might have a context object called `InsuredName` that extract the value `To.DisplayName`.

The `IDocumentTemplateDescriptor` interface is closely tied to the XML file format which corresponds to the default implementation of the `IDocumentTemplateSerializer` interface. The `IDocumentTemplateDescriptor` API consists mostly of property getters, with one setter (for `DateModified`) and some additional utility methods.

For locations of templates and descriptors, see “Template Source Reference Implementation” on page 223.

Template Descriptor Fields for Metadata About Each Template

The following list contains properties and methods that relate to metadata as defined in the `IDocumentTemplateDescriptor` interface. The second column specifies whether that property is required or optional within the XML files that the default implementation uses.

Property	Required in the XML file in the default implementation	Description
<code>templateId</code>	Required	The unique ID of the template
<code>name</code>	Required	A human-readable name for the template
<code>identifier</code>	Optional	An additional human-readable identifier for the template. This often corresponds to a well-known domain-specific document code. For example, to indicate a state-mandated form for this template.
<code>scope</code>	Required	The contexts that this template supports. Possible values are <ul style="list-style-type: none"> • <code>ui</code> – the document template must only be used from the document creation user interface • <code>rules</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>password</code>	Optional	If present, holds the password which is required for the user to create a document from the template. May not be supported by all document formats (for example, not supported for Gosu templates). In Microsoft Word, use the option to Protect Document... and select the Forms option. Then use the same password in the descriptor file as used to protect the document. To merge policy data into the form, PolicyCenter needs to unlock it, merge the data, and relock the form. PolicyCenter checks whether a form is locked and attempts to unlock it (and later relock it) using the password provided.
<code>mimeType</code>	Required	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> .
<code>documentProductionType</code>	Optional	If present, a specified document production type 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 <code>mimeType</code>).
<code>dateModified</code>	Optional	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
<code>dateEffective, dateExpiration</code>	Required	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

Property	Required in the XML file in the default implemen- tation	Description
keywords	Optional	A set of keywords search for within the template. Delimit keywords with the pipe () symbol in the XML file.
requiredPermission	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).
mailmergetype	Optional	Optional configuration of pagination of client-side Microsoft Word production. By default, PolicyCenter uses Microsoft Word <i>catalog pagination</i> , which correctly trims the extra blank page at the end. However, catalog pagination forbids template substitution in headers and footers. In contrast, <i>standard pagination</i> adds a blank page to the end of the file but enables template substitution in headers and footers. Set this attribute to the value <i>catalog</i> to use catalog pagination. To use standard pagination, do not set this attribute.

Methods in the IDocumentTemplateDescriptor interface related to metadata

getMetadataPropertyNames method	n/a	This method returns the set of extra metadata properties which exist in the document template definition. This 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.
getMetadataPropertyNames method	n/a	This method returns the set of extra metadata properties which exist in the document template definition. This 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.
getMetadataPropertyValue method	n/a	This method gets a property value. The method takes one argument, which is a property name as a String value. See getMetadataPropertyNames().

Template Descriptor Fields and Defaults for Document Metadata

The following template descriptor fields define the defaults for document type and security types:

Property	Required in the XML file in the default implemen- tation	Description
templateType / type (see note)	Required	Corresponds to the DocumentType typelist. Documents created from this template have their type fields set to this value. This is the only property in the XML file format with a property name difference from the property in the interface: <ul style="list-style-type: none"> • In the XML this property is called type • In the IDocumentTemplateDescriptor interface, this property is called templateType
defaultSecurityType	Required	Security type in the DocumentSecurityType typelist. This is the security type that becomes the default value for the corresponding document metadata fields for documents created using this template.

Template Descriptor Fields and Context Objects

As you write templates that include Gosu expressions, you need to reference business data such as the current Policy entity. Reference entities within templates as objects called *context objects*. Context objects create variables that form field expressions can refer to by name.

In addition to context objects that you define, form fields can always reference the policy using the `policy` symbol, for example, `policy.property` or `policy.methodName()`.

Only having access to one or two high-level root objects would get challenging. For example, suppose you want to address a policy acknowledgement letter to the main contact on the policy. Without context objects, each form field would have to repeat the same prefix (`Policy.MainContact.`) 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 could become hard to read with complex lengthy prefixes. You can simplify template code by creating a context object that refers to the intended recipient. Each context object must have two attributes: a unique name (such as "To") and a type (as a string, such as "Contact"; see the following discussion for the legal values). In addition, each context object must contain a `DefaultObjectValue` tag. This tag can contain any valid Gosu expression that identifies the default value to use for this `ContextObject`. You can construct a context object for your recipient as follows:

```
<ContextObject name="To" type="Contact">
<DefaultObjectValue>Policy.MainAccountContact</DefaultObjectValue>
</ContextObject>
```

You can simplify the form fields to the following:

```
<FormField name="ToName">To.DisplayName</FormField>
<FormField name="ToInsuredCity">To.PrimaryAddress.City</FormField>
<FormField name="ToState">To.PrimaryAddress.State</FormField>
<FormField name="ToZip">To.PrimaryAddress.PostalCode</FormField>
...

```

Context objects serve a second purpose by allowing you to manually specify the final value of each context object within the user interface. 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 document wizard can optionally allow users to pick from a list of possible values using the `PossibleObjectValues` tag as follows:

```
<ContextObject name="To" type="Contact">
  <DefaultObjectValue>Policy.MainAccountContact</DefaultObjectValue>
  <PossibleObjectValues>Policy.getMyCustomRelatedContactsExtensionMethod()</PossibleObjectValues>
</ContextObject>
```

The `PossibleObjectValues` value must be a Gosu expression that evaluates to an array, typically an array of Guidewire entities although that is not required. PolicyCenter does not enforce that the `DefaultObjectValue` is of the same type as `PossibleObjectValues`. While this makes it possible to have two different types for one context object, generally Guidewire recommends against this approach. If you were to do so, you must write form field expressions that work with two different types for that context object.

Context objects must be of one of the following types:

Context object type	Meaning
<code>EntityName</code>	The name of any PolicyCenter entity such as <code>Policy</code> , or <code>Activity</code> . The possible object values appear as a drop-down list of options. If that type of entity has a special type of picker, it also displays. 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>Entity</code>	This is a sort of "wildcard" type that indicates support for any keyable entity. Unlike the entry listed earlier with a specific entity name, this type is actually the literal string "Entity". This is useful for heterogeneous lists of objects.
<code>TypekeyName</code>	The name of any PolicyCenter typekey such as "YesNo".
<code>string</code>	A case-sensitive all-lower-case string to appear in the user interface as a single line of text. The <code><DefaultObjectValue></code> tag must be present, and its contents indicates the default text for this context object. Ignores the <code><PossibleObjectValues></code> tag.
<code>text</code>	Case sensitive, must be all lower case. 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.

By default, the `name` attribute of the context object becomes the user-visible name. If you want to use a different user-visible name, set the context object's `display-name` attribute to the text that you want to be visible to the user.

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 is of type `String`, then the user would typically be given a single-line text entry box.

If the context object has 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`.

The `IDocumentTemplateDescriptor` interface includes the following methods related to context objects:

- `String[] getContextObjectNames()` – Returns the set of context object names defined in the document template. See the XML document format for more information on what the underlying configuration looks like.
- `String 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` and `getFormFieldCompiledExpression` – In the rare case you need to implement the `IDocumentTemplateDescriptor` interface, these two methods require special return result types. For information about implementing these methods, contact Guidewire Customer Support.

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 of Built-in IDocumentTemplateSerializer

The default implementation of `IDocumentTemplateSerializer` uses an XML format that closely matches the fields in the `DocumentTemplateDescriptor` interface. This is intentional. The purpose of `IDocumentTemplateSerializer` is to serialize template descriptors and let you define the templates within simple XML files. The XML format is suitable in typical implementations. PolicyCenter optionally supports different potentially elaborate implementations that might directly interact with a document management system storing the template configuration information.

IMPORTANT Many of the fields in this section are defined in more detail in the previous section, “Document Template Descriptors” on page 213. The XML format described in this section is basically a serialization of the fields in the `IDocumentTemplateDescriptor` interface. In the reference implementation, the built-in `IDocumentTemplateDescriptor` and `IDocumentTemplateSerializer` classes implement the serialization.

The default implementation `IDocumentTemplateSerializer` is configured by a file named `document-template.xsd`. The default implementation uses XML that looks like the following:

```
<?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, 2003"
    date-expires="Apr 3, 2004"
    keywords="CA, reservation">
    <ContextObject name="To" type="Contact">
        <DefaultObjectValue>Policy.MainAccountContact</DefaultObjectValue>
        <PossibleObjectValues>Policy.getRelatedContacts()</PossibleObjectValues>
    </ContextObject>
    <ContextObject name="From" type="Contact">
        <DefaultObjectValue>Policy.AssignedUser.Contact</DefaultObjectValue>
        <PossibleObjectValues>Policy.getRelatedUserContacts()</PossibleObjectValues>
    </ContextObject>
    <ContextObject name="CC" type="Contact">
        <DefaultObjectValue>Policy.Driver</DefaultObjectValue>
        <PossibleObjectValues>Policy.getRelatedContacts()</PossibleObjectValues>
    </ContextObject>
    <FormFieldGroup name="main">
        <DisplayValues>
            <NullDisplayValue>No Contact Found</NullDisplayValue>
            <TrueDisplayValue>Yes</TrueDisplayValue>
            <FalseDisplayValue>No</FalseDisplayValue>
        </DisplayValues>
        <FormField name="PolicyNumber">Policy.PolicyNumber</FormField>
    </FormFieldGroup>
</DocumentTemplateDescriptor>
```

At run time, this XSD is referenced from a path relative to the module `config\resources\doctemplates` directory. If you want to change this value, in the plugin registry for this plugin interface, in 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" assuming you keep it in its original directory.

The attributes on the `DocumentTemplateDescriptor` element correspond to the properties on the `IDocumentTemplateDescriptor` API.

Date Formats in the Document Template XML File

Date values may be specified in the XML file in any of the following formats for systems in the English locale:

Date format	Example
MMM d, yyyy	Jun 3, 2005
MMMM d, yyyy Note: Four M characters mean the entire month name)	June 3, 2005
MM/dd/yy	10/30/06
MM/dd/yyyy	10/30/2006
MM/dd/yy hh:mm a	10/30/06 10:20 pm
yyyy-MM-dd HH:mm:ss.SSS	2005-06-09 15:25:56.845
yyyy-MM-dd HH:mm:ss	2005-06-09 15:25:56
yyyy-MM-dd'T'HH:mm:ss zzz	2005-06-09T15:25:56 -0700
EEE MMM dd HH:mm:ss zzz yyyy	Thu Jun 09 15:24:40 -0700 2005

Refer to the following codes for the date formats listed in the earlier table:

- a = AM or PM
- d = day
- E = Day in week (abbrev.)
- h = hour (24 hour clock)
- m = minute
- M = month (MMMM is entire month name)
- s = second
- S = fraction of a second
- T = parse as time (ISO8601)
- y = year
- z = Time Zone offset.

The first three formats typically are the most useful because templates typically expire at the end of a particular day rather than at a particular time. If you use any of the `template_tools` command line tools commands, you cannot rely on the date format in input files remaining. Although PolicyCenter preserves the values, the date format may change.

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.

Context Objects in the Document Template Descriptor XML File

A context object is an object attached to the merge request. You can insert a context object into the document or insert only certain fields within the object to the document.

For each `ContextObject` tag, the `DefaultObjectValues` expression determines the object (a contact in this case) to initially select. The `PossibleObjectValues` expression determines the set of objects to display in the select control.

Notice that all Gosu expression are in the contents of the tag, rather than attributes on the tag, which makes formatting issues somewhat easier. Also, there is still always a `Policy` entity called `policy` in context as the template runs. If you have time-intensive lookup operations, perform lookups once for the entire document, rather than once for each field that uses the results. To do this, write Gosu classes that implement the lookup logic and cache the lookup results as needed.

Form Fields in the Document Template Descriptor XML File

Form fields dictate a mapping between PolicyCenter data and merge fields in the document template. For example, to merge the policy number into the field `PolicyInfo`, use the following expression:

```
<FormField name="PolicyInfo">Policy.CorePolicyNumber</FormField>
```

`FormField` tags can contain any valid Gosu expression. This capability is most useful to combine with Studio-based utility libraries or class extensions. For example, to render the policy number and also other information about the policy, encapsulate that logic into an entity enhancement method called `Policy.getPolicyInformation()`. Reference that function in the form field as follows:

```
<FormField name="PolicyInfo">Policy.getPolicyInformation()</FormField>
```

Form fields can have two additional attributes: `prefix` and `suffix`. Both attributes are simple text values. Use these in cases in which you want to always display text such as “The policy information is ____”, so you can rewrite the form field to look like:

```
<FormField name="MainContactName"
  prefix="The policy information is "
  suffix=". ">
  Policy.getPolicyInformation()</FormField>
```

Form Fields Groups in the Document Template Descriptor XML File

You can optionally define form field groups that logically group a set of form fields. Groups are most useful for defining common attributes across the set of form fields, such as a common date string format. You can also use groups to define a `String` to display for the values `null`, `true`, or `false`. For example, you could check a Boolean value and display the text “`Police report was filed.`” and “`Police report was NOT filed.`” based on the results. Or, display a special message if a property is `null`. For example, for a doctor name field, display “`No Doctor`” if there is no doctor.

Form field groups are implemented as a `FormFieldGroup` element composed of one or more `FormField` elements. A descriptor file can have any number of `FormFieldGroup` elements. Typically, the Gosu in the `FormField` tags refer to a context object defined earlier in the file (see “Template Descriptor Fields and Context Objects” on page 216).

You can group display values for multiple fields by defining a `DisplayValues` tag within the `FormFieldGroup`. You can specify only one value (for example, `NullDisplayValue`) or some other smaller subset of values; you do not need to specify all possible display values. The possible choices are `NullDisplayValue` (if `null`), `TrueDisplayValue` (if `true`), and `FalseDisplay` (if `false`).

For Gosu expressions with subobjects that are pure entity path expressions and any object in the path evaluates to `null`, the expression silently evaluates to `null`. For example, if `Obj` has the value `null`, then `Obj.SubObject.Subsubobject` always evaluates to `null`. Consequently, the `NullDisplayValue` is a simple way of displaying something better if any part of the a multi-step field path expression is `null`.

PolicyCenter also uses the `NullDisplayValue` if an invalid array index is encountered. For example, the expression “`MyEntity.doctor[0].DisplayName`” results in displaying the `NullDisplayValue` if the `doctor` array is empty.

However, this approach does not work with method invocations on a `null` expression. For example, the expression `Obj.SubObject.Field1()` throws an exception if `Obj` is `null`.

In addition to checking for specific values, the `DisplayValues` tag can contain a `NumberFormat` tag, and either a `DateFormat` tag or a `TimeFormat` tag. These tags allow the user to specify a number format (such as “`###,##.##`”) or Date format (such as “`MM/dd/yyyy`”). The number formats modify the display of all form field values that are numbers or dates. This is useful in cases in which every number value in a form must be formatted in a particular way. Specify the format just once rather than repeatedly. Number format codes must contain the # symbol for each possible digit. Use any other character to separate the digits.

The `PolicyPeriodAPI` SOAP interface includes a method to add documents to a policy revision:

- `addDocumentToPolicyPeriod` – Add a new document to a policy revision

Customization of the XML Descriptor Mechanism

If the default XML-based template descriptor mechanism is used, the set of attributes can still be modified to suit your needs. 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 legal 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. See the “Using the PCF Editor” on page 305 in the *Configuration Guide* for more information about PCF configuration.

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.

Finally, once PolicyCenter creates the merged document, it tries to match attributes on the document template with properties on the document entity. For each match found, 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.

Template Source Reference Implementation

The PolicyCenter reference implementation keeps a set of document templates on the PolicyCenter application server. The document templates reside initially in the directory:

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

At run time, the document templates are typically at the path:

`SERVER/webapps/pc/modules/configuration/config/resources/doctemplates`

For each template, there are two files:

File	Naming	Example	Purpose
The template	Normal 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 + ".descriptor"	"Test.doc.descriptor"	This XML-formatted file provides various information about the template: <ul style="list-style-type: none">• How to search and find this template• Form fields that define what information populates 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.

See also

- For details on the XML formatting of template descriptor files, see:
 - “Document Template Descriptors” on page 213
 - “XML Format of Built-in IDocumentTemplateSerializer” on page 219.

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

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`.

Built-in Document Production Plugins

Each application uses a different mechanism for defining fields to be filled in by the document production plugins. The following table describes the built-in document production plugins and their associated formats:

Application or format	Description of built-in template
Microsoft Word	The document production plugin takes advantage of Microsoft Word's built-in mail merge functionality to generate a new document from a template and a descriptor file. PolicyCenter includes with a sample Reservation of Rights Word document and an associated CSV file. That CSV file is present so that you can manually open the Word template. PolicyCenter does not require or use this document.
Adobe Acrobat (PDF)	PDF file production happens on the server and requires a license to server-based software. See "Server-side PDF Licensing" on page 212.
Microsoft Excel	The document production plugin takes advantage of Microsoft Excel's built-in named fields functionality. The names 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	Gosu provides you with a sophisticated way to generate any kind of text file. This includes plain text, RTF, CSV, HTML, or any other text-based format. The document production plugin retrieves the template and uses the built-in Gosu language 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 resultant document in the appropriate application. See "Data Extraction Integration" on page 609.

Generating Documents from Gosu

PolicyCenter can generate PDF documents and Gosu-based forms without user intervention from Gosu business rules or from any other place Gosu runs.

PolicyCenter automatic form generation does not support any client-side creation such as Microsoft Word and Microsoft Excel templates. In client-side creation, the Document Assistant generates the new document locally. On Windows, the Document Assistant uses JScript to control local applications on the client PC. At the time business rules run, there may or may not be a user manipulating the application user interface. Because there may be no client to run the client-side code, client-side document production is impossible for automatic document creation.

However, server-side creation is possible using text formats:

- You can convert Word documents convert to the text-based RTF format
- You can convert many Excel templates to CSV files.

Therefore, you can convert most Microsoft application documents into equivalent Gosu templates that can generate text in RTF or CSV format. You can also use Gosu to generate any other text-based format, most notably plain text and HTML.

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.

The following list describes differences in automatic form generation compared to steps in "User Interface Flow for Document Production" on page 206. See that section for the step numbers:

- **Automatic template choosing** – Because template choosing is automatic, this effectively skips step 1 through step 6. Instead, you specify the appropriate template and parameter information within Gosu code.
- **No user editing** – Since there is no user intervention, there is no optional step for user intervention within the middle of this flow of step 9
- **No user uploading** – Because the Gosu code is executed on the PolicyCenter server, there is no user uploading step, which is step 11 in that section.

To create a document, first create a map of values that specifies the value for each context object. 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.

Within business rules, the Gosu class that handles document production is called `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`.

IMPORTANT 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 business 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.

See also

- “Important Notes About Cached Document UIDs” on page 226.

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:

```
// 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("RelatedTo", myPolicy)
parameters.put("Policy", myPolicy)

// Create and store the document using the context objects
DocumentProduction.createAndStoreDocumentSynchronously("EmailSent.gosu.htm", parameters, document)
```

This particular example assumes that the document production plugin for that template supports synchronous production. This is true for all built-in document production plugins but not necessarily for all document production plugins. The calling code must either know in advance whether the document production plugin for that template type supports synchronous and/or asynchronous creation, or checks beforehand. If necessary, you can check which types of production are supported with code such as:

```
var plugin : IDocumentProduction;  
plugin = PluginRegistry.getPluginRegistry().getPlugin(IDocumentProduction) as IDocumentProduction;  
if (plugin.synchronousCreationSupported(templateDescriptor)) {  
    ...  
}
```

For more examples of creating documents from Gosu, see “Document Creation” on page 105 in the *Rules Guide*.

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.

Template Web Service APIs

To manipulate Gosu document templates from external system, you can use the PolicyCenter `TemplateToolsAPI` web service. This web service is WS-I compliant.

Validating Templates

To validate templates, the `TemplateToolsAPI` web service provides several methods:

`validateTemplate` – Validate that the given template descriptor is in a valid format, and that all template descriptor context objects and form fields are valid given the current datamodel:

- Check that the Gosu expressions in the descriptor are valid. Context object default and possible value expressions must use the available objects. Form field expressions must use available objects or context objects.
- Check that the `permissionRequired` attribute, if specified, is a valid system permission code.
- Check that the `default-security-type` attribute, if specified, is a valid document security type code.
- Check that the `type` attribute, if specified, is a valid document type code.
- Check that the `section` attribute, if specified, is a valid section type code.

The method takes a template ID as a String value such as "ReservationRights.doc". The method returns human-readable string detailing the operations performed and any errors.

- `validateTemplateInLocale` – Same as `validateTemplate`, but specifies a specific locale code as an extra method argument.
- `validateAllTemplates` – validates all templates. The method returns human-readable string detailing the operations performed and any errors.
- `validateAllTemplatesInLocale` – Same as `validateAllTemplates`, but specifies a specific locale code as an extra method argument.
- `listTemplates` – List the templates which the server currently knows about. Useful for sanity-checking arguments to the validation commands. The method takes no arguments. It returns a single String value that lists all template names.
- `importFormFields` – Imports context objects, field groups, and fields from a comma-separated values (CSV) file into the corresponding template descriptor file. Arguments include:
 - `contextObjectFileContents` – The contents of a file containing the context objects to be imported, in CSV format.
 - `fieldGroupFileContents` – The contents of a file containing the field groups to be imported, in CSV format.
 - `fieldFileContents` – The contents of a file containing the fields to be imported, in CSV format.
 - `descriptorFileContents` – The contents of the descriptor file.

The method returns a results object that lists the fields for the new contents of the descriptor file, and a human-readable string detailing operations performed and any errors encountered.

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 the addresses are geocoded already.

This topic includes:

- “Geocoding Plugin Integration” on page 229
- “Steps to Deploy a Geocoding Plugin” on page 231
- “Writing a Geocoding Plugin” on page 232
- “Geocoding Status Codes” on page 237

See also

- For general information about implementing plugins, see “Plugin Overview” on page 123.

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 decides 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 let callers of the plugin determine whether the registered implementation supports routing services and reverse geocoding.

Note: The base configuration of PolicyCenter does not provide any user interface that uses 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 let you 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.

For more information, see “Reinsurance Integration” on page 403.

What the Geocoding Plugin Does

A `GeocodePlugin` implementation generally performs the following tasks:

- Assign latitude and longitude coordinates (required)
- List possible address matches (optional)
- Return driving directions (optional)
- Find an address from coordinates (optional)
- Find maps for arbitrary addresses (optional)

Synchronous and Asynchronous Calls to the Geocoding Plugin

From the perspective of PolicyCenter calling a `GeocodePlugin` method, the call 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 allows 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

- “Scheduling and Configuring Geocode Batch Processing” on page 20 in the *System Administration Guide*.

Using a Proxy Server with the Geocoding Plugin

If you want 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. It never responds to communications initiated external from the Internet. Therefore, you do not need a *reverse proxy server* to insulate the geocoding plugin from incoming Internet requests.

See also

- “Proxy Servers” on page 621.
- To learn how to configure the Bing Maps geocoding plugin to use a proxy server, see “Bing Maps Geocoding Service Communication” on page 623.

The 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. For details about configuring Bing Maps support, see “Configuring Geocoding” on page 19 in the *System Administration Guide*.

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` to filter which addresses to pass to the plugin for geocoding. If the property is `true`, they pass 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.

Steps to Deploy a Geocoding Plugin

Follow these high-level steps to deploy a `GeocodePlugin` implementation:

Step 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 PolicyCenter Studio. See “Writing a Geocoding Plugin” on page 232.

If your license for PolicyCenter includes use of Reinsurance Management and you want to support searching for nearby locations, you must repeat this step in `ContactManager Studio`.

Step 2: Register the Plugin Implementation in Studio

1. In the `Project` window in Studio, navigate to `Configuration` → `config` → `Plugins` → `registry`, and then open `GeocodePlugin.gwp`.
2. In the pane on the right, select the `Enabled` checkbox.
A dialog box informs you that editing the plugin creates a copy in the current module.
3. Click `Yes`.
4. In the `Class` text box, enter the name of the plugin implementation class that you want to use.
5. Edit the `Parameters` table to specify parameters and values that your plugin implementation requires.
Typically, a `GeocodePlugin` implementation has security parameters for connecting to the external geocoding service, such as a username and password.

Step 3: Enable the User Interface for Desired Geocoding Features

You must enable the user interface for the desired geocoding features within PolicyCenter by adjusting parameters in the `config.xml` file.

For PolicyCenter, modify the `UseGeocodingInPrimaryApp` parameter. This specifies whether PolicyCenter displays the user interface to search for nearby locations. For example:

```
<param name="UseGeocodingInPrimaryApp" value="true"/>
```

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 PolicyCenter Studio.

The high level features and related plugin methods of the `GeocodePlugin` interface are:

- **Geocode an address** – `geocodeAddressBestMatch`
- **List possible matches for an address** – `geocodeAddressWithCorrections`, `pluginSupportsCorrections`
- **Retrieve driving directions between two addresses** – `getDrivingDirections`,
`pluginSupportsDrivingDirections`, `pluginReturnsOverviewMapWithDrivingDirections`,
`pluginReturnsStepByStepMapsWithDrivingDirections`
- **Retrieve a map for an address** – `getMapForAddress`, `pluginSupportsMappingByAddress`
- **Retrieve an address from a pair of geocode coordinates** – `getAddressByGeocodeBestMatch`,
`pluginSupportsFindByGeocode`
- **List possible addresses from a pair of geocode coordinates** – `getAddressByGeocode`,
`pluginSupportsFindByGeocodeMultiple`

The `geocodeAddressBestMatch` method is the only method required of a `GeocodePlugin` implementation to be considered functional. The other methods are for optional features of the `GeocodePlugin`.

Using the Abstract Geocode Java Class

Although you can write your own implementation of the `GeocodePlugin`, Guidewire provides a built-in implementation of the plugin interface, called `AbstractGeocodePlugin`. It is an abstract Java class that your Gosu implementation can extend. The default behaviors of `AbstractGeocodePlugin` may save you work, particularly if you do not support all the optional features of the plugin. This built-in, abstract implementation is defined in the package `gw.api.geocode`.

If you use `AbstractGeocodePlugin` as the base class of your implementation, your Gosu class must provide implementations of these 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 {  
    Omit "implements GeocodePlugin" because AbstractGeocodePlugin already implements the interface.
```

2. Implement the required method `geocodeAddressBestMatch`.

The method accepts an address and returns a different address with latitude and longitude coordinates assigned.

See “Geocoding an Address” on page 233.

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`.

See “Getting Driving Directions” on page 235.

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. Be sure to let PolicyCenter know your plugin supports these features by implementing the methods that identify feature support.

For more information, see:

- “Supporting Multiple Address Corrections with a List of Possible Matches” on page 234
- “Retrieving Overview Maps” on page 235
- “Adding Segments of the Journey with Optional Maps” on page 236
- “Getting a Map for an Address” on page 237
- “Getting an Address from Coordinates (Reverse Geocoding)” on page 237

Geocoding an Address

The `GeocodePlugin` interface has one required method, `geocodeAddressBestMatch`. The method takes an address (`Address`) instance and returns a different address instance. The address returned from the plugin 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 with data returned from the geocoding service, or for any reason. Instead, make a copy of the incoming address by using the `clone()` method on it. 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.

See also

- “Geocoding Status Codes” on page 237.

Geocoding an Address from the User Interface

If PolicyCenter wants to geocode an address immediately, PolicyCenter calls one of the geocoding plugin methods. In situations in which PolicyCenter wants only the best match for a geocoding request, it calls the plugin method `geocodeAddressBestMatch`.

If you trigger geocoding from the user interface, geocoding is synchronous. In other words, the user interface blocks until the plugin returns the geocoding result. There is no built-in timeout between the application and the geocoding plugin. Your own geocoding plugin must encode a timeout so it can give up on the external service, throw a `RemoteException`, and let the user interface continue.

See also

- “Handling Address Clarifications for a Geocoded Address” on page 234

Geocoding an Address from a Batch Process

PolicyCenter geocodes addresses in the background with batch processes that call the geocoding plugin as necessary to geocode an address. For more information about how to configure the batch processes, See the “Working with Geocode Batch Processing” on page 19 in the *System Administration Guide*.

Handling Address Clarifications for a Geocoded Address

If `geocodeAddressBestMatch` is the only method from `AbstractGeocodePlugin` that you override, your plugin does really handle address correction. To support address correction, override the `geocodeAddressWithCorrections` and `pluginSupportsCorrections` methods. Also, you must implement a PCF to display multiple address and let the user select the correct one.

However, 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 the your plugin returns to match.

Callers of the plugin must assume that blank properties in a returned address are blank purposely. 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 returned by the geocoding plugin to be the full set of properties to show to the user or 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 a user can select the appropriate one. Some variations might be due to differences in abbreviations, such `Street` versus `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.

See also

- “Supporting Multiple Address Corrections with a List of Possible Matches” on page 234.

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 tell 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 letting a user 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.

See also

- “Handling Address Clarifications for a Geocoded Address” on page 234
- “Gosu Generics” on page 243 in the *Gosu Reference Guide*

Geocoding Error Handling

If your plugin implementation fails to connect to the external geocoding service, this method throws the exception `java.rmi.RemoteException`. Your implementation must never set the geocode status of an address to `none`. Instead, throw an exception if the error is retryable. For more information, see “Geocoding Status Codes” on page 237.

Getting Driving Directions

To support driving directions, implement these 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 in cases in which geocoding is enabled, you must edit the relevant PCF pages.

It is important to understand that PolicyCenter does not require that the driving directions request be handled by the same external service as geocoding requests. If desired, the plugin could contact different services for these types of requests.

This 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 illustrating the route between the two addresses are available, the method returns those also.

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 namespace. To create one, you have two options.

You can directly create a new driving directions object:

```
var dd = new DrivingDirections()
```

Alternatively, you can use a helper method to initialize the object:

```
uses gw.api.geocode.GeocodeUtils  
var dd = GeocodeUtils.createPreparedDrivingDirections(startAddress, endAddress, unitOfDist)
```

Note: The driving directions object is implemented as a Guidewire internal Java class, not a Guidewire entity. This distinction is important, because you cannot extend its data model.

Retrieving Overview Maps

If your geocoding or routing service supports overview maps, first implement the method `pluginReturnsOverviewMapWithDrivingDirections` to 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 (in other words, an HTML tag) that properly displays this map. This text may 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

For example:

```
dd.MapOverviewUrl = new MapImageUrl()
dd.MapOverviewUrl.MapImageUrl = "http://myserver/mapengine?lat=3.9&long=5.5"
```

Adding Segments of the Journey with Optional Maps

If you want to provide actual driving directions, you must also add driving directions elements that represent the segments of the journey.

Each `DrivingDirections` object contains various properties that relate to the entire journey, and it contains a list of journey segments in the form of an array of `DrivingDirectionsElem` objects. 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 earlier.

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 may represent either a discrete stage in the directions, such as “Turn left onto I-80 for 10 miles”. It may represent a note or milestone not corresponding to a stage, such as “Start trip”. The exact format and content of the textual description depends greatly on your geocoding service. It may or may not 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 chronological order for the journey. In other words, generate the segments in the expected order from the start address to the end address.

If you add individual maps like this, also implement the method

```
pluginReturnsStepByStepMapsWithDrivingDirections
```

to return `true`.

Extracting Data from Driving Directions in PCF Files

If you want to extract information from `DrivingDirections` in PCF code or other Gosu code, be aware 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 3D 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`.

Refer to the Gosu API Reference in Studio for the full set of properties.

Error Handling

If the plugin fails to connect to the external geocoding service, throw the exception `java.rmi.RemoteException`.

Getting a Map for an Address

Some geocoding services support getting a map from an address. If your service supports it, first implement the `pluginSupportsMappingByAddress` method and return true to tell PolicyCenter that your implementation supports this feature. Next, implement the method `getMapForAddress`, which takes an `Address` and a unit of distance (miles or kilometers).

Your `getMapForAddress` method must return a map image URL in a `MapImageUrl` object. This is a simple wrapper object containing a `String` for a map URL.

The following 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 getting an address from latitude and longitude coordinates. This is sometimes called *reverse geocoding*. 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 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 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 geocoding multiple-result methods, this returns a list of `Address` entities, specified with the generics syntax `List<Address>`. For more information, see “Gosu Generics” on page 243 in the *Gosu Reference Guide*.

Geocoding Status Codes

Geocoding services typically provide a set of status codes to indicate what happened during the geocoding attempt. For example, 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. Do they represent an exact address match? If the service could not find the address or the address was incomplete, do the coordinates 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. That means that any values in the `Latitude` and `Longitude` properties of the address are 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.

The status values must be values from the GeocodeStatus type list, described in the following table:

Geocode status code	Description
none	No attempt at geocoding this address occurred. This is the default geocoding status for an address. Do not set an address to this geocoding status. If you experience an error that must retrigger geocoding later, throw an exception instead. When an address is modified, PolicyCenter sets the address to this status, too, which indicates that the address has not been geocoded since it was last modified.
failure	An attempt at geocoding this address was made but failed completely. If an address could not be geocoded, use this code. The geocoding batch process retries the failed address. Do not use this code for an error that is retryable, such as a network failure. If you experience an error that must retrigger geocoding later, throw an exception instead.
exact	This address was geocoded successfully, and the service provider indicates that the supplied geocode coordinates match exactly the complete address.
street	This address was geocoded successfully, and the service provider indicates that the supplied geocode coordinates match the street but not the complete address. This can be more or less precise than postalcode, depending on the complete address and the length of the street.
postalcode	This address was geocoded successfully, and the service provider indicates that the supplied geocode coordinates match the postal code but not the complete address. The meaning of a postal code match depends on the geocoding service. A geocoding service may use the geographically weighted center of the area, or it may use a designated address within the area, such as a postal office.
city	This address was geocoded successfully, and the service provider indicates that the supplied geocode coordinates match the city but not the complete address. The meaning of a city match depends on the geocoding service. A geocoding service may use the geographically weighted center of the city, or it may 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

You can store certain data model properties encrypted in the database. For example, you can hide important data, such as bank account numbers or private personal data, by storing the data in the database in a non-plaintext format. This topic is about how to integrate PolicyCenter with your own encryption code.

This topic includes:

- “Encryption Integration Overview” on page 239
- “Changing Your Encryption Algorithm Later” on page 244

Encryption Integration Overview

You can store certain data model properties encrypted in the database. For example, hide important data, such as bank account numbers or private personal data, in the database in a non-plaintext format. The actual encryption and decryption is implemented through the `IEncryption` plugin. You can define your own algorithm. PolicyCenter does not provide an encryption algorithm in the product. PolicyCenter simply calls this plugin implementation, which is responsible for encoding an unencrypted `String` or reversing that process.

IMPORTANT PolicyCenter provides a sample encryption algorithm that simply reverses the string value. You must disable this plugin if you do not wish to use encryption. You must implement the plugin to properly encrypt your data with your own algorithm.

All encryption and decryption is done within PolicyCenter automatically as the application accesses and loads entities from the database. All Gosu rules and web services automatically operate on plain text (unencrypted text) without special coding to support encryption within rules, web services, and messaging plugins.

From Gosu, any encrypted fields appear unencrypted. If you must avoid sending this information as plain text to external systems, you must design your own integrations to use a secure protocol or encrypt that data.

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 (for example, SSL/HTTPS) or is on a secure network, additional encryption might not be necessary. It depends on the details of your security requirements.

WARNING From Gosu, any encrypted fields appear unencrypted. Carefully consider security implications of any integrations with external systems for properties that normally are encrypted.

For communication between Guidewire 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

IMPORTANT The built-in integrations between Guidewire applications do not encrypt properties in the web service connection between the applications. To add additional security, you must customize the integration to use HTTPS or add additional encryption of properties sent across the network.

Setting Encrypted Properties

You can change the encryption settings for a column in data model files by overriding the column information (adding a *column override*). You can only set encryption for character-based column types such as `varchar`. Encryption is unsupported on other types, including binary types (`varbinary`) and date types.

There are two ways to mark the column as encrypted:

- Make a column encrypted by specifying its type as `encrypted`:

```
<column name="encrypted_column" type="encrypted" size="10"/>
```

This makes the column `encrypted_column` have the type `EncryptedString`. PolicyCenter generates visible masks for all input widgets for this value.

- Alternatively, mark a column as encrypted through the `encryption` attribute, which only applies to types based on `varchar`. This masks values that are accessed directly as entity properties. For example:

```
<column name="encrypted_column" type="varchar" size="10" encryption="true"/>
```

This example makes the property `encrypted_column` to be the type `String`. In this case, input widgets for this value are masked only if the value is a entity property access. For example, suppose the earlier column is within an entity. PolicyCenter generates an input mask if the value expression is the form `entity.encrypted_column`. However, there is no input mask if the expression is a method invocation like `pageHelper.getMyEncryptedColumnValue()`.

PolicyCenter prevents you from encrypting properties with a denormalized column. In other words, no encryption on a property that creates a secondary column to support case-insensitive search. For example, the contact property `LastnameDenorm` is a denormalized column added to mirror the `Lastname` property, and thus the `Lastname` property cannot be encrypted.

Querying Encrypted Properties

Gosu database query builder queries (and older-style `find` expressions) work as expected if you compare the property with a literal value. For example, looking up a social security number or other unique ID is comparing with a literal value. However, the encrypted data for every record is not decrypted from the database to test the results of the query. It actually works in the opposite way. If you compare a constant value against an encrypted property, PolicyCenter encrypts the constant in the SQL query. The query embeds the equality comparison logic directly into the SQL.

For comparison logic, the SQL that Gosu generates always compares either:

- Two properties from the database, either both encrypted or both unencrypted
- A static literal, encrypted if necessary, and a property from the database.

One side effect of this is that equality comparison is the only supported comparison for encrypted properties. For example you cannot use “greater than” comparisons or “less than” comparisons.

This means that your `find` queries can compare encrypted properties against other values in only two ways:

- Direct equality comparison (`==`) or not equals (`!=`) a Gosu expression of type `String`
- Direct equality comparison (`==`) or not equals (`!=`) with a field path to another encrypted property

For example, suppose `Policy.SecretVal` and `Policy.SecretVal2` are encrypted properties, suppose `Policy.OtherVal` and `Policy.OtherVal2` are unencrypted properties, and suppose `tempValue` is a local variable containing a `String`.

The following queries work:

```
q.compare("SecretVal", Equals, "123")
q.compare("SecretVal", NotEquals, "123")
q.compare("SecretVal", Equals, tempValue)
q.compare("SecretVal", NotEquals, tempValue)
q.compare("SecretVal", Equals, q.getColumnRef("SecretVal2")) // both encrypted
q.compare("SecretVal", NotEquals, q.getColumnRef("SecretVal2")) // both encrypted
q.compare("OtherVal", Equals, q.getColumnRef("OtherVal2")) // both unencrypted
q.compare("OtherVal", NotEquals, q.getColumnRef("OtherVal2")) // both unencrypted
```

The following queries do not work:

```
q.compare("SecretVal", Equals, q.getColumnRef("OtherVal")) // bad mix of encrypted and unencrypted
q.compare("SecretVal", NotEquals, q.getColumnRef("OtherVal")) // bad mix of encrypted and unencrypted
q.compare("SecretVal", GreaterThan, "123") // no greater than or less than
```

IMPORTANT If a Gosu `find` query involves encrypted properties, you can use equality comparison with other encrypted properties or against `String` values constants in the query. However, there are limitations with what you can do in the query. Refer to the instructions in this section for details.

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.

To encrypt, implement an `encrypt` method that takes an unencrypted `String` and returns an encrypted `String`, which may or may not be a different length than the original text. If you want to use strong encryption and are permitted to do so legally, you are free to do so. PolicyCenter does not include any actual encryption algorithm.

To decrypt, implement a `decrypt` method that takes an encrypted `String` and returns the original unencrypted `String`.

You must also specify the maximum length of the encrypted string by implementing the `getEncryptedLength` method. Its argument is the length of 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, then this method helps PolicyCenter know 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. This identifier tracks encryption-related change control. This is exposed to Gosu as the property `EncryptionIdentifier`:

```
override property get EncryptionIdentifier() : String {
    return "ABC:DES3"
}
```

The encryption ID is very important and 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 encryption ID of the current encryption plugin
- The encryption ID associated with the database last time the server ran.

For important details, see “[Changing Your Encryption Algorithm Later](#)” on page 244.

IMPORTANT Be careful that your encryption plugin returns an appropriate encryption ID and that it is unique among all your encryption plugins.

The following example is a simple demonstration encryption plugin. It simple fake encryption algorithm is to append a reversed version of the unencrypted text to the end of the text. For example, encrypting the text “hello” becomes “helloolleh”.

To decrypt the text, it merely removes the second half of the string. The second half of the string is the reversed part of the text that it appended earlier when encrypting the string. It is important to note that this fake algorithm doubles the size of the encrypted data, hence the `getEncryptedLength` doubles and returns the input size argument.

The following Gosu code implements this plugin

```
package Plugins
uses gw.plugin.util.IEncryption
uses java.lang.StringBuilder

class EncryptionByReversePlugin implements IEncryption {

    override function encrypt(value:String) : String {
        return reverse(value)
    }

    override function decrypt(value:String) : String {
        return value.subString(value.length() / 2, value.length());
    }

    override function getEncryptedLength(size:int) : int {
        return size * 2 // encrypting doubles the size
    }

    override property get EncryptionIdentifier() : String {
        return "mycompany:reverse"
    }

    private function reverse(value:String) : String {
        var buffer = new StringBuilder(value)
        return buffer.reverse().append(value).toString()
    }
}
```

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 authoritatively detect whether the property data is already encrypted.

For example, suppose you put a known series of special unusual characters that could not appear in the data both before and after your encrypted data. You can now detect whether data is encrypted or unencrypted:

- During an encryption request, if your encryption plugin detects that the data is already encrypted, throw an exception.
- During a decryption request, if your encryption plugin detects that the data is already decrypted, throw an exception.

Installing Your Encryption Plugin

After you write your implementation of the `IEncryption` plugin, you must register your plugin implementation. You can register multiple `IEncryption` plugin implementations if you need to support changing the encryption algorithm.

To enable your encryption plugin implementation

1. 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.

IMPORTANT Guidewire strongly recommends 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 on `registry`, and choose `New` → `Plugin`.
4. Studio prompts you to name your new plugin. Remember that you can have than one registered implementation of an `IEncryption` plugin interface. The name field must be unique among all your plugins. This is called the *plugin name* and is particularly important for encryption plugins.

IMPORTANT Guidewire strongly recommends you set the plugin name for your current encryption plugin to a name that describes the algorithm itself. For example, `encryptDES3`.

5. In the `interface` field, type `IEncryption`.
6. Edit standard plugin fields in the Plugins editor in Studio. See “Using the Plugins Registry Editor” on page 131 in the *Configuration Guide*. Also see “Plugin Overview” on page 123.
7. In `config.xml`, set the `CurrentEncryptionPlugin` parameter to the plugin name.

The `CurrentEncryptionPlugin` parameter specifies which encryption plugin is the current encryption algorithm for the main database. Specify the plugin name, not the class name nor the encryption ID.

WARNING 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 upgrader detects data model fields marked as encrypted but the database contains unencrypted versions, the upgrader encrypts the field in the main database using the current encryption plugin.

See also

“Changing Your Encryption Algorithm Later” on page 244

Adding or Removing Encrypted Properties

If you later add or remove encrypted properties in the data model, upgrader automatically runs on server startup to update the main database to the new data model.

Set Encryption Plugin for Encrypted Properties in Archived Objects

The `config.xml` configuration parameter `DefaultXmlExportIEncryptionId` specifies the unique encryption ID of an encryption plugin.

If archiving is enabled, the application uses that encryption plugin to encrypt any encrypted fields during XML serialization. For more about encryption unique IDs, see “Writing Your Encryption Plugin” on page 241.

WARNING To avoid accidentally archiving encrypted properties as unencrypted data, be sure to set the parameter `DefaultXmlExportIEncryptionId`.

Changing Your Encryption Algorithm Later

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.

Note: When you use the Plugins editor in Studio to add an implementation of `IEncryption`, Studio prompts you for a text value to use as the plugin name for this implementation. Guidewire strongly recommends you set the plugin name for encryption plugins to names that describe the algorithm. For example, “`encryptDES3`” or “`encryptRSA128`”. Any legacy encryption plugins (if you did not originally enter a name) have the name “`IEncryption`”.

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`.

The most important things to remember whenever you change encryption algorithms are:

- 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.

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, assure every encryption plugin returns its appropriate encryption ID, and assure `CurrentEncryptionPlugin` specifies the correct plugin name.

Be careful not to confuse the encryption ID of a plugin implementation with its plugin name or 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.

Changing Your Encryption Algorithm

The following procedure describes how to change your encryption algorithm. It is extremely important to follow it exactly and very carefully. If you have questions about this before doing it, contact Guidewire Professional Services before proceeding.

WARNING Do not follow this procedure until you are sure you understand it and test your encryption algorithm code. Before proceeding, be confident of your encryption code, particularly your implementation of the plugin method `getEncryptedLength`. Failure to perform this procedure correctly risks data corruption.

1. Shut down your server.
2. Register a new plugin implementation of the `IEncryption` plugin for your new algorithm. When you add an implementation of the plugin in Studio, it prompts you for a plugin name for your new implementation. Name it appropriately to match the algorithm. For example, "encryptDES3".
3. Be sure your plugin returns an appropriate and unique encryption ID. Name it appropriately to match the algorithm. For example, "encryptDES3".
4. Set the `config.xml` configuration parameter `CurrentEncryptionPlugin` to the plugin name of your new encryption plugin.
5. Start the server. The upgrader uses the old encryption plugin to decrypt your data and then encrypts it with the new algorithm.

See also

- “Writing Your Encryption Plugin” on page 241

Critical Warning: If You Change Encryption Algorithms, You May Still Need the Old Algorithm

If you change your encryption algorithm, it is important to realize that you may need to continue to register your encryption plugin implementations for your older algorithm.

For archived policies, changes to encryption do not occur immediately in the main database as part of the normal upgrader. Even after running the upgrader, you must continue to register encryption plugin implementations for your old encryption algorithms. The archive data includes the encryption ID for the plugin that originally encrypted the data. When PolicyCenter restores that policy, the application needs the encryption plugin for older algorithms to decrypt encrypted properties. Next, PolicyCenter uses the current encryption algorithm to encrypt any encrypted properties.

WARNING Register one encryption plugin implementation for every encryption algorithm that might be referenced by encryption ID in any archived data. Ensure that all encryption plugin implementations return the correct encryption ID.

Management Integration

This topic discusses *management*, which is a special type of API that allows external code to control PolicyCenter in certain ways through the JMX protocol or some similar protocol. Types of control include viewing or changing cache sizes and modifying configuration parameters dynamically. This topic focuses on writing a new management plugin.

This topic includes:

- “Management Integration Overview” on page 247
- “The Abstract Management Plugin Interface” on page 248
- “Integrating With the Included JMX Management Plugin” on page 249

See also

- For more information about plugins, see “Plugin Overview” on page 123.
- For the complete list of all PolicyCenter plugins, see “Summary of All PolicyCenter Plugins” on page 141.

Management Integration Overview

PolicyCenter provides APIs to view and change some PolicyCenter settings from remote management consoles or APIs from remote integration code:

- **Configuration parameters** – View the values of all configuration parameters, and change the value of certain parameters. These changes take effect in the server immediately, without requiring the server to restart. After the server shuts down, PolicyCenter discards dynamic changes like this. After the server starts next, the server rereads parameters in the `config.xml` configuration file.
- **Batch processes** – View batch processes currently running (if any).
- **Users** – View the number of current user sessions and their user names
- **Database connections** – View the number of active and idle database connections
- **Notifications** – View notifications about locking out users due to excessive login failures.

PolicyCenter exposes this data through three mechanisms:

- **User interface in Server Tools tab, on the Management Beans page** – All of these items are exposed in the user interface on the Management Beans page of the PolicyCenter Server Tools tab. This tab is accessible only to users with the soapadmin permission. For more information, see “Management Beans” on page 162 in the *System Administration Guide*.
- **Abstract management plugin interface** – PolicyCenter also provides an abstract interface called the *management plugin*. This plugin exposes *management beans* to the external world. Management beans are interfaces to read or control system settings. For example, the management plugin could expose the management beans using the Java Management Extension (JMX) protocol, using the Simple Network Management Protocol (SNMP), or something else entirely.
- **A supported JMX management plugin for Apache Tomcat** – PolicyCenter includes an example plugin that implements JMX using the management plugin. This plugin is compiled and automatically installed in `PolicyCenter/modules/configuration/plugins` so that you need only to enable it in Studio. The included JMX management plugin works only with the Apache Tomcat web server. The source code for this example is included in at this path:

`PolicyCenter\java-api\examples\src\examples\plugins\management`

The Abstract Management Plugin Interface

The following table lists the important interfaces and classes used by the abstract management plugin interface.

Interface or class	Description
<code>ManagementPlugin</code>	The management interface for Guidewire systems. This is the interface that the included JMX plugin implements.
<code>GWMBean</code>	The interface for managed beans exposed by the system. The management plugin must expose these internal management beans to the outside world using JMX, SNMP, or some other management interface.
<code>GWMBeanInfo</code>	Meta-information about a Guidewire managed bean (<code>GWMBean</code>), such as its name and description.
<code>ManagementAuthorizationCallbackHandler</code>	A callback handler interface that PolicyCenter invokes if a user attempts management operations.
<code>NotificationSenderMarker</code>	A marker interface that indicates that a <code>GWMBean</code> can send notifications.
<code>Attribute</code>	The <code>Attribute</code> class encapsulates two strings that represent a system attribute name and its value.
<code>AttributeInfo</code>	Metadata about an <code>Attribute</code> , such as name, description, type, and whether the attribute is readable and/or writable.
<code>Notification</code>	A PolicyCenter notification, such as those sent if PolicyCenter locks out a user due to too many failed login attempts.
<code>NotificationInfo</code>	Metadata about a notification, such as a message string, a sequence number, and a notification type.

The main task of the management plugin is to register `GWMBean` objects. Registering the object means that the plugin provides that service to the outside world in whatever way makes sense for that service. For example, the management plugin might create a wrapper for the `GWMBean` and expose it using JMX or SNMP using its own published service.

For details, refer to these Java source files in the directory `PolicyCenter/java-api/examples`:

- `JMXManagementPlugin.java`
- `GWMBeanWrapper.java`
- `JMXAuthenticatorImpl.java`
- `JSR160Connector.java`

See also

- To write code for an external system to communicate with the built-in JMX management plugin, see “Integrating With the Included JMX Management Plugin” on page 249.

Integrating With the Included JMX Management Plugin

PolicyCenter includes an example plugin that implements JMX using the management plugin interface. This plugin is compiled and automatically installed in `PolicyCenter/modules/configuration/plugins` so that you need only to change one setting in the Plugins Registry editor in Studio to enable it.

In the Project window in Studio, select **Configuration** → **config** → **Plugins** → **registry**, and then open `ManagementPlugin.gwp`. Select the **Enabled** checkbox. This supported JMX management plugin works only with the Apache Tomcat web server. The source code for this example is included as part of the PolicyCenter examples directory, at the following path:

```
PolicyCenter\java-api\examples\src\examples\plugins\management
```

Note: JMX support differs between web servers, but the JMX management plugin is designed for and supported in Guidewire production environments that use Apache Tomcat only. The JMX management plugin is supported for jetty in development environments. Modifications to the source code and use of different JMX libraries could enable the management plugin for other web servers. Guidewire provides the source code in case such changes are desired.

After you enable the JMX plugin, it publishes a JMX service called a *JMX JSR160 connector* under Apache Tomcat. A remote management console or your integration code can call the JMX JSR160 connector by creating *JMX connector client* code that connects to the published service.

The following example demonstrates a simple Java application that can retrieve a system attribute programmatically from a remote system. This example relies on the following conditions:

- The PolicyCenter server must be running under the Apache Tomcat web server.
- The JMX plugin must be installed (which it is by default).
- The JMX plugin must be enabled in the Plugins Registry editor

This example retrieves the `HolidayList` system attribute programmatically from a remote system:

```
package com.mycompany.pc.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;

// Create a "JMX JSR160 connector" to connect to PolicyCenter
//
public class TestJMXClientConnector {

    public static void main(String[] args) {

        try {
            // The address of the connector server
            JMXServiceURL address = new JMXServiceURL(
                "service:jmx:rmi:///jndi/rmi://akitio:1099/jrmp");

            // The creation environment map, null in this case
            Map creationEnvironment = null;

            // Create the JMXConnectorServer
            JMXConnector cctor = JMXConnectorFactory.newJMXConnector(address,
                creationEnvironment);
        }
    }
}
```

```
// May contain - for example - user's credentials
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
cntor.connect(environment);

// Obtain a "stub" for 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");
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.

This topic includes:

- “Territory Code Plugin” on page 251
- “Vehicle Identification Number Plugin” on page 253
- “Automatic Address Completion and Fill-in Plugin” on page 253
- “Phone Number Normalizer Plugin” on page 254
- “Testing Clock Plugin (Only For Non-Production Servers)” on page 254
- “Work Item Priority Plugin” on page 255
- “Official IDs Mapped to Tax IDs Plugin” on page 255
- “Preupdate Handler Plugin” on page 256
- “Defining Base URLs for Fully-Qualified Domain Names” on page 256
- “Exception and Escalation Plugins” on page 257

See also

- For general information about plugins, see “Plugin Overview” on page 123.
- For messaging plugins, see “Messaging and Events” on page 289.
- For authentication plugins, see “Authentication Integration” on page 181.
- For document and form plugins, see “Document Management” on page 191.
- For geocoding plugins, see “Geographic Data Integration” on page 229.

Territory Code Plugin

PolicyCenter tracks territories, which are geographic regions, for each line of business. Each territory must have a unique territory code that identifies the region and the line of business. PolicyCenter supports two different usage models for territories and how to load them into PolicyCenter.

The system table data is stored in the entity DB_Territory. The entity called Territory is an extension that is a non-persisted entity consisting only of a foreign key to a DB_territory entity.

Use the territory code plugin (`TerritoryCodePlugin`) to help map territory search criteria (city, state, postal code, and policy line type) and return an array of Territory entities. This plugin could consult an external system to do the lookup.

If you want territories to work very differently, you can modify the data model for the (new) Territory entity type in PolicyCenter to be anything you want. Just be sure to modify the territory code plugin accordingly with any changes including custom fields. This would allow you to look up territory data directly from the plugin. The plugin could call out using a web service to an external system, thus you would not need to import territory data locally.

Default System Table Approach

In the default approach, you can load Territory entities as part of the PolicyCenter product model reference data as a system table. You can modify the file:

```
PolicyCenter/modules/configuration/config/resources/productmodel/systables/territory_codes.xml
```

This file loads into the local PolicyCenter database if loading the product model. Populate this file with territory codes, each of which looks like:

```
<Territory public-id="demo_sample:25">
  <City>San Mateo</City>
  <Code>AAA</Code>
  <County>San Mateo</County>
  <Description/>
  <EffectiveDate>2000-01-01 00:00:00.000</EffectiveDate>
  <ExpirationDate/>
  <PolicyLinePattern public-id="BusinessAutoLine"/>
  <PostalCode>99999</PostalCode>
  <State>CA</State>
</Territory>
```

Typically, this data is derived from government agency data that manages the territory codes.

To use this data, simply use the built-in implementation of the territory plugin, `TerritoryCodeDemoPlugin`, which is registered for you in the default `config.xml` file:

```
<plugin-java name="ITerritoryCodePlugin"
  javaclass="com.guidewire.pc.plugin.territorycode.internal.TerritoryCodeDemoPlugin"/>
```

This plugin simply searches for Territory entities in an internal system table created from the product model XML file mentioned earlier.

In the alternative approach, design a new plugin implementation that fetches Territory entities from an external system or performs some other dynamic calculation of territory codes. For example, if you have a legacy system that stores this information centrally, the territory plugin could query that system. In that case, PolicyCenter would ignore the territory system table created from the product model XML file mentioned earlier.

The territory code plugin interface (`ITerritoryCodePlugin`) contains only one method (`getTerritoryCode`). This method takes a territory lookup criteria (`TerritoryLookupCriteria`) object. This lookup criteria object includes information like the city, state, postal code, and policy line type. The method must populate and return an array of Territory entities returned from the external system. If there is a single match on the search, return an array of length one containing a single Territory.

If there are multiple results, return an array with more items. If you return multiple results, PolicyCenter provides a user interface to select a territory code among the results. You can go to the Locations page, open a location, and select a territory code. If this plugin returns multiple results, the order of the items within the array are irrelevant to the user interface. Make the number of returned codes a small number. Smaller numbers create a better user experience. Your plugin is responsible for limiting the number of results returned to a manageable number.

Return an empty array if no territory codes can be found. Throw an exception if there is an error somewhere, and PolicyCenter displays a user interface based on the type of exception.

The following is an example of this plugin. It returns the territory associated with the passed-in search criteria. * The territory contains a code and a physical place. The criteria is required to have non-null values for state, policy line pattern and effective date. In the default configuration, the territory has a foreign key to the database entity DBTerritory, which is a system table.

```
public class TerritoryCodePlugin implements ITerritoryCodePlugin {  
  
    override function getTerritoryCodes(criteria: TerritoryLookupCriteria): Territory[] {  
        if (criteria.State == null) throwError(displaykey.Java.TerritoryCodePlugin.Error.MissingState)  
        if (criteria.PolicyLinePattern == null)  
            throwError(displaykey.Java.TerritoryCodePlugin.Error.MissingPolicyLinePattern)  
        if (criteria.EffectiveOnDate == null)  
            throwError(displaykey.Java.TerritoryCodePlugin.Error.MissingEffectiveOnDate)  
        var query = PCDependencies.getTerritoryFinder().findTerritoriesByCriteria(criteria)  
        var territories = new ArrayList<Territory>()  
        var iterator = query.iterator()  
        while (iterator.hasNext()) {  
            territories.add(new Territory(criteria) {DBTerritory = iterator.next()})  
        }  
        return territories as Territory[]  
    }  
  
    private static function throwError(message : String) {  
        throw new DisplayableException(message)  
    }  
}
```

If you do not want to use the system table, you have two options:

1. Change the plugin implementation to get territory information from an other entity type that stores this information rather than system tables.
2. Reconfigure the Territory entity to have additional relevant properties and populate those with values from an external system.

Vehicle Identification Number Plugin

The `IVinPlugin` plugin is a simple plugin that returns vehicle information from a standard vehicle identification number (a so-called *VIN number*). The one 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. This 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.

See also

- “Address Autocompletion and Autofill” on page 144 in the *Globalization Guide*
- “Configuring Zone Information” on page 126 in the *Globalization Guide*

Phone Number Normalizer Plugin

To support automatic address completion, implement the `IPhoneNormalizerPlugin` plugin interface.

There is a default implementation called `DefaultPhoneNormalizerPlugin`, which handles the default behavior.

Contact Guidewire Customer Support for more information.

See also

- “DefaultNANPACountryCode” on page 60 in the *Configuration Guide*

Testing Clock Plugin (Only For Non-Production Servers)

WARNING The `ITestingClock` plugin is supported only for testing on non-production development servers. Do not register this plugin on production servers.

Testing PolicyCenter behavior over a long span of time during multiple billing cycles can be challenging. For testing on development servers only, you can implement the `ITestingClock` plugin and programmatically change the system time to simulate the passing of time. For example, you can define a plugin that returns the real time except in special cases in which you artificially increase the time to represent a time delay. The delay could be one week, one month, or one year.

This plugin interface has only 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 temporarily cannot be reached, throw the exception `java.lang.IllegalArgumentException`.

Time must always increase, not go back in time. Going back in time is likely to cause unpredictable behavior in PolicyCenter.

Notes:

- The plugin method `setCurrentTime` advances the time only for `gw.api.util.DateUtil.currentTimeMillis`. The class `java.util.Date` always states the server time. Therefore, you must use `gw.api.util.DateUtil.currentTimeMillis` in your implementation code.
- The advanced date does not change in all parts of the product. There are certain areas that intentionally do not use the rolled-over date. For example, the time shown in the next scheduled run for batch processes is not affected.
- Today’s date on the `Calendar` does not change.

Using the Testing Clock Plugin

The base application has an implementation of the `ITestingClock` plugin interface. This topic shows you how to use it.

1. Start PolicyCenter Studio.
2. In the Project window, navigate to `configuration` → `config` → `Plugins` → `registry`.
3. Right-click `registry` and click `New` → `New Plugin`.
4. In the `Name` field, type `ITestingClock`.
5. In the `Interface` field, type `ITestingClock`.
6. Click `OK`.

7. In the Plugins Registry editor, click **Add plugin** +, and then choose Java Plugin.
8. In the **Java class** field, type the following:
`com.guidewire.pl.plugin.system.internal.OffsetTestingClock`
9. In the Project window, navigate to **configuration** → **config**, and then open **config.xml**.
10. In **config.xml** under Environment Parameters, 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 -->
<param name="EnableInternalDebugTools" value="true"/>`
11. Save your changes.
12. Stop then restart PolicyCenter.
13. Log in as user **su** with password **gw**.
14. 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 as needed:
Run Clock addDays 10
In the yellow message area near the top of the screen, a message shows the new current date.
15. 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**.

Testing Clock Plugin in PolicyCenter Clusters

If you are operating a cluster of PolicyCenter servers, you must use the following procedure to change the time.

To change the testing clock time for PolicyCenter clusters

1. Stop all servers with the exception of the *batch server*.
2. Advance the testing clock.
3. Restart all the cluster nodes.

Work Item Priority Plugin

Configure how PolicyCenter calculates work item priority for workflow steps by implementing the Work Item Priority plugin (`IWorkItemPriorityPlugin`). The interface has a single method, `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.

IMPORTANT Prioritization affects only work items of type `WorkflowWorkItem` or its derivatives.

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.

Preupdate Handler Plugin

To implement preupdate handling, register an implementation of the `IPreUpdateHandler` plugin interface.

Your plugin implementation must implement one method, `executePreUpdate`. This method takes a single preupdate context object, `PreUpdateContext`, and returns nothing.

The `PreUpdateContext` object has several properties you can get, which return a list (`java.util.List`) of entities in the current database transaction.

From Gosu they look like the following properties:

- `InsertedBeans` – An unordered list of entities added in this transaction.
- `UpdatedBeans` – An unordered list of entities changed in this transaction.
- `RemovedBeans` – An unordered list of entities added in this transaction.

From Java, these properties appear as three methods: `getInsertedBeans`, `getUpdatedBeans`, and `getRemovedBeans`.

Default Plugin Implementation

In the base configuration, PolicyCenter provides the plugin implementation `gw.plugin.preupdate.impl.PreUpdateHandlerImpl`.

By default, this plugin implementation class collects all inserted and updated beans, as well as accounts and jobs if any assignments changed.

Any exception cause the bounding database transaction to roll back, effectively undoing the update.

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 base URL builder plugin (`IBaseURLBuilder`) plugin and registering it with the system.

You can base your implementation on the built-in example implementation found at the 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.

Implement `IBaseURLBuilder` and `InitializablePlugin`

Your `IBaseURLBuilder` plugin must explicitly implement `InitializablePlugin` in the class definition. Otherwise, PolicyCenter does not initialize your plugin.

For example, suppose your plugin implementation's first line looks like this:

```
class MyURLBuilder implements IBaseURLBuilder {
```

Change it to this:

```
class MyURLBuilder implements IBaseURLBuilder, InitializablePlugin {
```

To conform to the new interface, you must also implement a `setParameters` method even if you do not need parameters from the plugin registry:

```
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
}
```

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 Studio if you want something other than the default behavior.

Use these 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	IActivityEscalationPlugin	Calls the activity escalation rule set
Group exceptions	IGroupExceptionPlugin	Calls the group exception rule set
User exceptions	IUserExceptionPlugin	Calls the user exception rule set

Startable Plugins

A *startable plugin* is a special type of code that runs without human intervention in the application server as a background process, beginning at server startup.

This topic includes:

- “Startable Plugins Overview” on page 259
- “Writing a Startable Plugin” on page 260
- “Configuring Startable Plugins to Run on All Servers” on page 263
- “Java and Startable Plugins” on page 266
- “Persistence and Startable Plugins” on page 266

See also

- For an alternative mechanism to startable plugins, see “Developing Custom Batch Processes” on page 582.

Startable Plugins Overview

You can register custom code that runs at server startup in the form of a startable plugin implementation. You can use a startable plugin as a daemon, such as listener to a JMS queue. You can use a startable plugin for periodic batch processing, such as deleting expired files from a file system folder. You can use a startable plugin as an initializer, such as loading configuration data into the application database.

You can start and stop startable plugins as circumstances require. You cannot start or stop standard types of plugins. Instead, code in a standard Guidewire plugin executes only when other code invokes its methods.

Startable Plugins as Background Processes

Typically, a startable plugin behaves as a daemon, such as a listener that runs continuously as a background process. Startable plugins have similarities with the messaging reply plugin. A messaging transport's main task is to send a message to an external system and listen for the reply acknowledgment. This metaphor might not apply to integrations that do not need to send outgoing messages about data changes using the event and messaging system. However, if you need listener code and it must initialize with server startup, use a startable plugin.

Note: Some integrations require external code to trigger an action within PolicyCenter but do not need custom listener code. Depending on your needs, consider publishing a PolicyCenter web service instead of a startable plugin. For more information, see “Web Services Introduction” on page 37.

Registering Startable Plugins

Register your startable plugin in the Plugins Registry in Studio, just like you register standard plugins. In the Project window, navigate to **configuration** → **config** → **Plugins** → **registry**. All startable plugin implementations that you register in the Plugins Registry and that are enabled are listed on the **Server Tools** → **Startable Plugin** page.

See also

- For details on how to register a startable plugin, see “Working with Plugins” on page 132 in the *Configuration Guide*.

Starting, Stopping, and Managing Startable Plugins

If you have administration privileges, you can view the operational status of registered startable plugins on the **Server Tools** → **Startable Plugin** page. In the Actions column, use the **Start** and **Stop** buttons to start and stop startable plugins. You can modify the PCF files for the **Startable Plugin** page to show additional information about your startable plugins, their underlying transport mechanisms, or any other information.

See also

- For information on using a web service to start and stop startable plugins, see “Stopping Startable Plugins Using Web Services” on page 97.

Startable Plugins in a Clustered Configuration

By default for a cluster, startable plugins operate only on the batch server. However, you can configure a startable plugin to run on all servers in the cluster.

See also

- “Configuring Startable Plugins to Run on All Servers” on page 263

Writing a Startable Plugin

Most of your code of a startable plugin implements your custom listener code or other special service. Write a new class that implements the **IStartablePlugin** interface and 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 variable definition with the property definition can be combined with the single variable definition:

```
var _state : StartablePluginState as State
```

The plugin does include a constructor called on system startup. However, start your service code in the `start` method, not the constructor. Your start method must appropriately set the state (started or stopped) using an internal variable that you define.

At minimum, your start method must set your internal variable that tracks your started or stopped state, with code such as:

```
_state = Started
```

Additionally, in your `start` method, start any listener code or threads such as a JMS queue listener.

Your `start` method has a method parameter that is a callback handler of type `StartablePluginCallbackHandler`. This callback is important if your startable plugin modifies any entity data, which is likely the case for most real-world startable plugins.

Your plugin must run any code that affects entity data within a code block that you pass to the callback handler method called `execute`. The `execute` method takes as its argument a *Gosu block*, which is a special type of in-line function. The Gosu block you pass to the `execute` method takes no arguments. For more information, see “Gosu Blocks” on page 235 in the *Gosu Reference Guide*.

Note: 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. To run your code as a specific user, see “User Contexts for Startable Plugins” on page 262.

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 `Transaction.getCurrent()` to get the current bundle if you need a reference to the current (writable) bundle. For more information, see “Bundles and Database Transactions” on page 337 in the *Gosu Reference Guide*.

IMPORTANT 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. For more information, see “Defining Startable Plugins In Java” on page 266.

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 `Server Tools` 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. Refer to “Updating Entity Instances in Query Results” on page 174 in the *Gosu Reference Guide* for more information about bundles.

This simple 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 ) : void  
{  
    _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."
}

//Note: 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. For more information, see “Adding Entity Instances to Bundles” on page 341 in the *Gosu Reference Guide*.

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 `Server Tools` 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 preupdate 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. The method signature is:

```
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 is a complete simple startable plugin. This 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

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 ) : void
{
    _callback = cbh
    _callback.execute( \ -> {
        // Do some work:
        // [...]
    } )
    _state = Started
    _callback.log( "**** From HelloWorldStartablePlugin: Hello world." )
}

override function stop( isShuttingDown: boolean ) : void
{
    _callback.log( "**** From HelloWorldStartablePlugin: Goodbye." )
    _callback = null
    _state = Stopped
}
```

Startable Plugins and Run Levels

On server startup, by default PolicyCenter starts all startable plugins when the server gets to the `maintenance` run level, which is also called `nodaemons` in some APIs.

If you want your startable plugin to start at an earlier or later run level, add the annotation `@gw.api.server.Availability` on the plugin implementation class declaration. Pass a run level (`gw.api.server.AvailabilityLevel`) as an argument to the annotation constructor.

For example:

```
uses gw.api.server.Availability
uses gw.api.server.AvailabilityLevel

@Availability(AvailabilityLevel.MULTIUSER)
class HelloWorldStartablePlugin implements IStartablePlugin {

    ...
}
```

This annotation is meaningful for startable plugins but not other types of plugins. Other types of plugins are instantiated and initialized on first usage, independent of the run level.

Configuring Startable Plugins to Run on All Servers

By default for a cluster, startable plugins run only on the batch server. However, you can configure them to run on all servers in the cluster by declaring the startable plugin is *distributed*. To declare the plugin is distributed, add the `@Distributed` annotation to the startable plugin implementation class.

Guidewire strongly recommends that startable plugins very clearly and consistently save the state (started or stopped) across all servers in a cluster. This approach handles edge cases like a server joining the cluster late after other servers have started. To keep the state consistent across all servers, the state must persist in the database. Even though the state information in the database is not automatically persisted, the broadcast to other servers happens automatically.

The series of steps is as follows:

1. On any server, the server starts. The plugin method reads the second argument to the `start` method. Because the server is starting up, this value is `true`. The plugin implementation detects this condition and reads the database status of the started or stopped state and sets its own state appropriately.
2. Later, the state may change on a server to start or stop from the user interface or through APIs. The application propagates this information to all servers in the cluster. Each server in the cluster decides whether it is appropriate to respect this request:

- If the startable plugin is distributed, the server respects the request and calls the plugin `start` or `stop` method as appropriate.
- If the startable plugin is not distributed, only the batch server handles the request.

However, if the startable plugin is not distributed and the original changed server is the batch server, the behavior is different. The batch server does not bother with the distributed broadcast system because it is the only running version of the startable plugin. Instead, that server immediately calls the plugin `start` or `stop` method as appropriate.

3. On any server, the server shuts down. The plugin method reads the second argument to the `stop` method. Because the server is shutting down, this value is `true`. The plugin implementation detects this condition and simply stops its local state but does not set the database status in the database.

Implement getting and setting the state information in the database using the plugin callback handler object, which is an instance of `StartablePluginCallbackHandler`. This object is a parameter to the `start` method of the plugin. Save a copy of this object as a private variable and then call the following methods on the object as needed:

- `getState` – Gets the state information in the database: `true` for started, `false` for stopped. If this is the first time in history that this startable plugin has ever run on this server-database cluster combination, its state information is not in the database. This method takes one arguments that represents the default assumption to use the very first time this query is checked. For example, if you expect this startable plugin to always run, pass the value `Started`. If this startable plugin runs only in rare situations and you expect only an administrator or API to trigger it to start, pass the value `Stopped`. If the server state has ever been set in the database, `getState` ignores this argument.
- `setState` – Sets the state information in the database for this startable plugin. This method takes one argument, which is true if and only if the startable plugin is running. This request attempts to set this information in the database if it is not yet set to the expected value. If the startable plugin state is already matching the database value for the state according to the `getState` method, do not call `setState`. Be careful to catch any exceptions. You must create your own bundle to make this change, see the example for how to use the `runWithNewBundle` API.
- `logStart` – Writes a line in the log about the plugin starting. This method takes one argument, which is line to log, as a `String`. Include the plugin name and the state, such as `"HelloWorldDistributedStartablePlugin:Started"`.
- `logStop` – Writes a line in the log about the plugin stopping. This method takes one argument, which is line to log, as a `String`. Include the plugin name and the state, such as `"HelloWorldDistributedStartablePlugin:Stopped"`.

The following is an example implementation of the `start` and `stop` methods that implement a trivial thread and the proper setting of startable plugin 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
uses gw.api.util.Logger

@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 {
        _callback = handler
        if (serverStartup) {
            // if the server is starting up, read the value from the database, and default to stopped
        }
    }
}
```

```
// if this plugin NEVER saved its state before. You might want to change this to Started instead.
_state = _callback.getState(Stopped)

if (_state == Started) {
    _callback.logStart("HelloWorldDistributedStartablePlugin:Started")
}
else {
    _callback.logStart("HelloWorldDistributedStartablePlugin:Stopped")
}
else {

    _state = Started
    if (_callback.State != Started) {
        changeState(Started) // call our internal function that sets the database state if necessary
    }
    _callback.logStart("HelloWorldDistributedStartablePlugin")
}

// if the thread already existed for some reason, briefly stop it before starting it again
if (_state == Started) {
    if (_thread != null) {
        _thread.stop()
    }
    _startedHowManyTimes++

// create your thread and start it
var t = new Thread() {
    function run() {
        print("hello!")
    }
}
t.Daemon=true

}

override function stop( serverStopping : boolean ) : void {
    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) // call our internal function that sets the database state if necessary
            }
            _callback.logStop("HelloWorldDistributedStartablePlugin")
        }
    }
    _state = Stopped
}

// our internal function that sets the database state if necessary. if there are exceptions,
// this implementation tries 5 times. You might want a different behavior.
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)
        }
    }
}
```

Java and Startable Plugins

You can develop your custom startable plugin in Java, but special considerations apply.

Defining Startable Plugins In Java

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`.

For example:

```
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);
```

For information about Gosu blocks and inner classes, see “Gosu Block Shortcut for Anonymous Classes Implementing an Interface” on page 211 in the *Gosu Reference Guide*.

Location of Java Files for Startable Plugins

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.

The instructions are slightly different depending on whether you define the plugin interface implementation itself in Java or in Gosu:

- If your main startable plugin class is a Java class, see “Special Notes For Java Plugins” on page 130.
- If your main startable plugin class is a Gosu class, see “If Your Gosu Plugin Needs Java Classes and Library Files” on page 130.

Persistence and Startable Plugins

Your startable plugin can manipulate Guidewire entity data. If your startable plugin needs to maintain state for itself, do one of the following:

- 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 requests. PolicyCenter includes default implementations for the most common usages: reading text file data and receiving JMS messages.

This topic includes:

- “Multi-threaded Inbound Integration Overview” on page 267
- “Inbound Integration Configuration XML File” on page 269
- “Inbound File Integration” on page 271
- “Inbound JMS Integration” on page 276
- “Custom Inbound Integrations” on page 278
- “Understanding the Polling Interval and Throttle Interval” on page 285

Multi-threaded Inbound Integration Overview

There are sometimes situations that require high-performance data throughput for inbound integrations that require special threading or transaction features from the hosting J2EE/JEE application environment. It is difficult to interact with the application server’s transactional facilities and write correct, thread-safe, high-performing code. PolicyCenter includes tools that help you write such inbound integrations. You can focus on your own business logic rather than how to write thread-safe code that works safely in each application server.

Inbound Integration Configuration XML File

There is a configuration file for inbound integrations called `inbound-integration-config.xml`. Edit this file in Studio to define configuration settings for every inbound integration that you want to use. Each inbound integration requires configuration parameters such as references to the registered plugin implementations. See “Inbound Integration Configuration XML File” on page 269.

Inbound Integration Core Plugin Interfaces

PolicyCenter provides the multi-threaded inbound integration system in two different plugin interfaces for different use cases. Each defines a contract between PolicyCenter and inbound integration high-performance multi-threaded processing of input data:

- `IInboundIntegrationMessageReply` – Inbound high-performance multi-threaded processing of replies to messages sent by a `MessageTransport` implementation. This is a subinterface of `MessageReply`.
- `IInboundIntegrationStartablePlugin` – Inbound high-performance multi-threaded processing of input data as a startable plugin. A startable plugin is for all contexts other than handling replies to messages sent by a message transport (`MessageTransport`) implementation. This is a subinterface of `IStartablePlugin`.

You might not need to write your own implementation of these main plugin interfaces. PolicyCenter includes plugin implementations that are supported for production servers that support common use cases. Both are provided in variants for message reply and startable plugin use.

Type of input data	Description	Related topic
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. See “Inbound File Integration” on page 271. 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 <i>handler</i> plugin.	“Inbound File Integration” on page 271
JMS inbound integration	Use this integration to get objects from a JMS message queue. See “Inbound JMS Integration” on page 276. You provide your own code that processes the next message on the JMS message queue. Your code is called a <i>handler</i> plugin.	“Inbound JMS Integration” on page 276
Custom integration	If you process incoming data other than files or JMS messages, write your own version of the <code>IInboundIntegrationMessageReply</code> or <code>IInboundIntegrationStartablePlugin</code> plugin interface. In both cases, for custom integrations you must write multiple classes that implement helper interfaces such as <code>WorkAgent</code> .	“Custom Inbound Integrations” on page 278

IMPORTANT You can implement your plugin code in any method you choose: Gosu, Java (with no OSGi), or in OSGi (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. See “Inbound Integration Handlers for File and JMS Integrations” on page 268.

When registering a plugin implementation in the Plugins Registry, you must add a plugin parameter called `integrationservice`. That `integrationservice` parameter defines how PolicyCenter finds configuration information within the inbound integration configuration XML file.

To configure the inbound integration configuration XML file, see “Inbound Integration Configuration XML File” on page 269.

For general information about the Plugins Registry, see “Registering a Plugin Implementation Class” on page 126.

Inbound Integration Handlers for File and JMS Integrations

Whether you use the built-in integrations or write your own, you must write code that handles one chunk of data.

To write a custom integration that supports data other than files or JMS messages, your code primarily implements the interface `InboundIntegrationMessageReply` or `InboundIntegrationStartablePlugin`. For custom integrations, “Custom Inbound Integrations” on page 278 and skip the rest of this topic.

In contrast, if you use the built-in file or JMS inbound integrations, you register a built-in plugin implementation of `InboundIntegrationMessageReply` or `InboundIntegrationStartablePlugin`, depending on whether you are using messaging. PolicyCenter includes plugin implementations of those interfaces that know how to process files or JMS data, depending on which one you choose.

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`. That method handles one chunk of data that PolicyCenter passes as a method of type `java.lang.Object`. Write a handler class that implements the `process` method. Downcast the `Object` to the necessary type:

- for file handling, downcast to `java.nio.file.Path` or a `String`, depending how you configure the integration to process by files or by line
- for JMS handling, downcast to a JMS message of type `javax.jms.Message`

Next, register the plugin in the Studio Plugins Registry. See “Registering a Plugin Implementation Class” on page 126.

IMPORTANT When registering your plugin implementation, you must also add one plugin parameter called `integrationService`. That plugin parameter links your plugin implementation to one XML element within the `inbound-integration-config.xml` file. See “Inbound Integration Configuration XML File” on page 269.

If you are using either the file or JMS integrations as the startable plugin variant, your class must implement the `InboundIntegrationHandlerPlugin` interface.

If you are using either the file or JMS integrations as the message reply variant, your class must implement the `InboundIntegrationMessageReplyHandler` interface. This is a subinterface of `InboundIntegrationHandlerPlugin`. Implement the basic `process` method as well as all methods of the `MessageReply` plugin. For example, implement `MessageReply` methods `initTools`, `suspend`, `shutdown`, `resume`, and `setDestinationID`. Save the parameters to your `initTools` method into private variables. Use those private variables during your `process` method find and acknowledge the original `Message` object. For related information, see “Implementing a Message Reply Plugin” on page 336.

You can implement your plugin code in any method you choose: Gosu, Java (with no OSGi), or in OSGi (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.

Inbound Integration Configuration XML File

In Studio, there is a configuration file for inbound integrations. In the Project window, navigate to `configuration` → `config` → `integration`, and the open `inbound-integration-config.xml`.

The file contains the following types of data:

- “Thread Pool Configuration” on page 269
- “Configuring a List of Inbound Integrations” on page 270

Thread Pool Configuration

The first section of the `inbound-integration-config.xml` file configures thread pools. Within the `<threadpools>` element, there is a list of `<threadpool>` elements, each of which look like the following

```
<threadpool name="gw_default" disabled="false">
  <gwthreadpooltype>FORKJOIN</gwthreadpooltype>
</threadpool>
```

The `name` attribute is a symbolic name that is used later in the XML file to refer uniquely to this thread pool.

The `disabled` attribute is a Boolean value that defines whether to disable that thread pool. If set to `true`, the thread pool is disabled.

Within the element, there are two supported element types for setting thread pool parameters:

- `<gwthreadpooltype>` – The thread pool type with the following supported values, which are case-sensitive:
 - Set to `FORKJOIN` for a self-managing default thread pool.
 - Set to `COMMONJ` for running WebSphere or WebLogic and defining the JNDI name of the thread pool. If you set this value, you must also set the `<workmanagerjndi>` parameter.
- `<workmanagerjndi>` – JNDI name of the thread pool. This parameter is required if the thread pool type is set to `COMMONJ`.

IMPORTANT In the default configuration, there are thread pools predefined for default WebSphere or WebLogic thread pools. These are for development but not production use. 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 `<threadpool>` settings to include the JNDI name for your new thread pool.

Configuring a List of Inbound Integrations

The second section of the `inbound-integration-config.xml` file contains a list of inbound integrations that you want to use. For example, if you want five different JMS inbound integrations, each listening to its own JMS queue, add five elements to this section of the file.

Within the `<integrations>` element, there is a list of subelements of several pre-defined element names. For each inbound integration, define configuration parameters as subelements. For example, you must declare the name of the registered plugin implementations that corresponds to your handler code.

Some of the parameters are required, and some are optional. If you use either the built-in file processing or JMS integration, there are special parameters just for those integrations.

The following table lists the supported integration element names and where to find the complete reference for parameter names.

Type of integration	Configuration element	For more information
File inbound integration	<code><file-integration></code>	"Inbound File Integration" on page 271
JMS inbound integration	<code><jms-integration></code>	"Inbound JMS Integration" on page 276
Custom inbound integration. Directly implement the <code>InboundIntegrationStartablePlugin</code> or <code>InboundIntegrationMessageReply</code> interface.	<code><custom-integration></code>	"Custom Inbound Integrations" on page 278

Your configuration XML element must have the following two attributes:

- `name` – A unique identifying name for this inbound integration in the `inbound-integration-config.xml` file.

IMPORTANT The `name` attribute must match the value of the `integrationservice` plugin parameter in the Plugins registry for all registered plugin implementations of any inbound integration interfaces. In the Plugins registry, add the plugin parameter `integrationservice` and set to the value of this unique identifying name. See "Registering a Plugin Implementation Class" on page 126 and "Using the Plugins Registry Editor" on page 131 in the *Configuration Guide*.

- `disabled` – Determines whether to disable this inbound integration. Set to `true` to disable the inbound integration. Otherwise, set to `false`.
- `env` – Sets a configurations that is valid only for a specific server environment. See “Varying Inbound Integration Settings” on page 271.

Varying Inbound Integration Settings

There are multiple ways you can vary configuration of inbound integration by system environment:

- **Within the inbound integration XML file** – Within your `inbound-integration-config.xml` file, each top-level element can have an `env` attribute. That attribute specifies that element is valid only for one value of the `env` system environment configuration setting. See “Defining the Application Server Environment” on page 14 in the *System Administration Guide*. 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. See “Configuring a List of Inbound Integrations” on page 270. Within a single Plugins Registry file, you can optionally define the `integrationservice` plugin property multiple times with values that vary by system environment (`env`) or server ID (`serverid`) values. See “Adding an Implementation to a Plugins Registry Item” on page 132 in the *Configuration Guide*.

You can use one or both of these techniques to vary the run time behavior of the server based on the server environment.

Note that within the elements in the `inbound-integration-config.xml` file, the `env` attribute is supported but you cannot vary the configuration by server ID. To do configuration by server ID, use the plugin property technique.

Inbound File Integration

PolicyCenter includes built-in code that supports file-based input with high-performance multi-threaded processing. PolicyCenter provides this code in two variants, one for processing message replies, and one as a startable plugin.

You cannot modify the plugin implementation code in Studio, but you can use one or more instances of these integrations to work with your own file data.

The processing flow for file integration is as follows:

1. In response to some inbound event, your own integration code creates a new file in a specified incoming directory on the local file system.
2. The inbound file integration code polls the *incoming directory* at a specified interval and detects any new files since the last time checked.

IMPORTANT Any files that are in the incoming directory before the plugin initializes are never processed. For example, if you restart the server, files created after the server shuts down but before initialization are never processed. To process files that already existed, wait until the plugin is initialized, then move files outside the *incoming directory*, and then back to the incoming directory again.

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 code moves all found files to the *processing directory*, which stores inbound files in progress.
 4. The inbound file integration code opens each new file using a specified character set. The default is UTF-8, but it is configurable.
 5. The inbound file integration code reads one unit of work (one chunk of data) and dispatches it to your handler code. The `processingmode` parameter in the `inbound-integration-config.xml` file defines the type of data processed in one unit of work. If that parameter has the value `line`, PolicyCenter sends one line at a time to the handler as a `String` object. If that parameter has the value `file`, PolicyCenter sends the entire file to the handler as a `java.nio.file.Path` object.
- If exceptions occur during processing, the plugin code moves the file to the *error directory*. The file name is changed to add a prefix that includes the time of the error, as 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`.
6. After successfully reading and processing the complete file, the inbound file integration code 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 code 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` parameter, mentioned later in this section.
 8. The inbound file integration waits until the next polling interval, and repeats this process at step 3.

To create an inbound file integration

1. In the Project window, navigate to `configuration` → `config` → `integration`, and the open `inbound-integration-config.xml`.
2. Configure the thread pools. See “Thread Pool Configuration” on page 269.
3. In the list of integrations, create one `<integration>` element of type `<file-integration>`. Follow the pattern in the file to set the XML element namespace. In the base configuration, type the element name as `<ci:i:file-integration>`. Set the `name` and `disabled` attributes as described in “Configuring a List of Inbound Integrations” on page 270.
4. Set configuration parameter subelements as follows:

File integration configuration parameters	Required	Description	Example value
<code>pluginhandler</code>	Required	The name in the Plugins registry for an implementation of the <code>InboundIntegrationHandlerPlugin</code> plugin interface. Note that this is the <code>.gwp</code> file name, not the implementation class name.	<code>InboundFileIntegrationExample</code>
<code>processingmode</code>	Required	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> .	<code>line</code>
<code>threadpool</code>	Required	The unique name of a thread pool as configured earlier in the file. See “Thread Pool Configuration” on page 269.	<code>gw_default</code>

File integration configuration parameters	Required	Description	Example value
osgiservice	Required	Always set to <code>true</code> . This is for internal use. This value is independent of whether you choose to register your handler class as an OSGi plugin.	<code>true</code>
transactional	Required	You must always set this to <code>false</code> .	<code>false</code>
createdirectories	Optional	If <code>true</code> , PolicyCenter creates the incoming, processing, error, and done directories if they do not already exist. If errors that prevent creation of any directories, the server does not startup. If <code>false</code> , PolicyCenter all of these directories must already exist. If any directories do not already exist, the server does not startup. For better security, set to <code>false</code> . The default is <code>false</code> .	<code>false</code>
stoponerror	Required	If <code>true</code> , PolicyCenter stops the integration if an error occurs. Otherwise, PolicyCenter just skips that item. Be sure to log any errors or notify an administrator.	<code>true</code>
incoming	Required	The full path of the configured incoming events directory	<code>/tmp/inbound/incoming</code>
processing	Required	The full path of the configured processing events directory	<code>/tmp/inbound/processing</code>
done	Required	The full path of the configured done events directory	<code>/tmp/inbound/done</code>
error	Required	The full path of the configured error events directory	<code>/tmp/inbound/error</code>
pollinginterval	Optional	The time interval in seconds between polls, though the algorithm interacts with the throttle interval and the ordered parameter. See "Understanding the Polling Interval and Throttle Interval" on page 285. The default is 60 seconds.	15

File integration configuration parameters	Required	Description	Example value
throttleinterval	Optional	The time interval in seconds after polling, though the algorithm interacts with the polling interval and the ordered parameter. See “Understanding the Polling Interval and Throttle Interval” on page 285. The default is 60 seconds.	15
ordered	Optional	<p>By default, the inbound file integration handles multiple files at a time in parallel in multiple server threads. If you want files handled in a single thread sequentially, set this value to true. The default is false.</p> <p>WARNING: The order of the processing of individual files in the directory is undefined. Never rely on the order being any particular order, such as alphabetic or creation date.</p> <p>Also see “Understanding the Polling Interval and Throttle Interval” on page 285.</p>	false

5. In Studio, within the Plugins registry, add a new .gwp file. For related information, see “Registering a Plugin Implementation Class” on page 126 and “Using the Plugins Registry Editor” on page 131 in the *Configuration Guide*.
6. 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:
 - For a message reply plugin, type `InboundIntegrationMessageReply`.
 - For a startable plugin for non-messaging use, type `InboundIntegrationStartablePlugin`.
7. Click the plus (+) symbol to add a plugin implementation and choose **Add Java plugin**.
8. In the **Java class** field, type:
 - 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`.
9. 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.
10. 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.InboundIntegrationHandlerPlugin`.
 - For a startable plugin for non-messaging use, your handler code must implement the plugin interface `gw.plugin.integration.inbound.InboundIntegrationMessageReply`.

You can implement your plugin code in any method you choose: Gosu, Java (with no OSGi), or in OSGi (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.

This interface has one method called `process`, which has a single argument of type `Object`. The method returns no value. The file integration calls that method to process one chunk of data. 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` before using it.
- If you set the `processingmode` parameter to `file`, downcast the `Object` to `java.nio.file.Path` before using it.

If your code throws an exception, PolicyCenter moves the file to the error directory. Note the `stoponerror` parameter in the configuration file. If that value is `true`, PolicyCenter stops the integration if an error occurs. Otherwise, skips that item. Be sure to log any errors or notify an administrator.

In Studio, within the Plugins registry, register your handler plugin implementation class. Add a row in the Plugins registry editor for your plugin implementation class. See “Registering a Plugin Implementation Class” on page 126 and “Using the Plugins Registry Editor” on page 131 in the *Configuration Guide*. Set the interface name to the handler interface that you implemented (see step 10).

IMPORTANT Within the Plugins registry for your handler plugin implementation, the `Name` field must match the `pluginhandler` parameter you use in the `inbound-integration-config.xml` file for this integration.

11. 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.
12. Start the server and test your new inbound integration.

Example File Integration

The following is an example file integration configuration in the `inbound-integration-config.xml` file:

```
<cii:file-integration name="exampleFileIntegration" disabled="true">
    <pluginhandler>InboundFileIntegrationHandler</pluginhandler>
    <pollinginterval>1</pollinginterval>
    <throttleinterval>5</throttleinterval>
    <threadpool>gw_default</threadpool>
    <ordered>true</ordered>
    <stoponerror>false</stoponerror>
    <transactional>false</transactional>
    <osgiservice>true</osgiservice>
    <processingmode>line</processingmode>
    <incoming>/tmp/incoming</incoming>
    <processing>/tmp/processing</processing>
    <error>/tmp/error</error>
    <done>/tmp/done</done>
    <charset>UTF-8</charset>
    <createdirectories>true</createdirectories>
```

The following is a simple handler class called `mycompany.integration.SimpleFileIntegration`, which prints the lines in the file:

```
package mycompany.integration
uses gw.plugin.integration.inbound.InboundIntegrationHandlerPlugin
class SimpleFileIntegration implements InboundIntegrationHandlerPlugin {
    // this example assumes the inbound-integration-config.xml file
    // sets this 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} (!)");
    }
}
```

For this example, in the Plugins Registry there are two plugin implementations in the Plugins registry:

- `InboundFileIntegration.gwp` – Registers an implementation of `InboundFileIntegrationPlugin` with the required Java class `com.guidewire.pl.integration.inbound.file.DefaultFileInboundIntegrationPlugin`. The `integrationservice` plugin parameter is set to `exampleFileIntegration`.
- `InboundFileIntegrationHandler.gwp` – Registers an implementation of `InboundIntegrationHandlerPlugin` with the Gosu class `mycompany.integration.SimpleFileIntegration`. The `integrationservice` plugin parameter is set to `exampleFileIntegration`.

When you start up the server, you will see the log line in the console for startup:

```
***** SimpleFileIntegration startup
```

After the server starts, add files to the `/tmp/incoming` directory. You will see additional lines for each processed line. As mentioned earlier, do not add files to the directory until after the plugin is initialized. Files that are in the incoming directory on startup are never processed.

Inbound JMS Integration

PolicyCenter includes a built-in high-performance multi-threaded integration with inbound queues of Java Message Service (JMS) messages. The inbound JMS integration supports application servers that implement the `commonj.work.WorkManager` interface specification. These currently include the IBM WebSphere and Oracle Weblogic application servers. Define your own code that processes an individual message, and the inbound JMS framework handles message dispatch and thread management.

PolicyCenter can use JMS implementations on the application server but PolicyCenter does not include its own JMS implementation. For additional advice on setting up or configuring an inbound JMS integration with PolicyCenter, contact Guidewire Customer Support.

To create an inbound JMS integration

1. In the Project window, navigate to `configuration` → `config` → `integration`, and then open `inbound-integration-config.xml`
2. Configure the thread pools. See “Thread Pool Configuration” on page 269.
3. In the list of integrations, create one `<integration>` element of type `<jms-integration>`. Follow the pattern in the file to set the XML element namespace. In the base configuration, type the element name as `<ci:i:jms-integration>`. Set the `name` and `disabled` attributes as described in “Configuring a List of Inbound Integrations” on page 270.
4. Set configuration parameter subelements as follows:

Plugin parameters in Plugins editor, description	Required	Description	Example value
<code>pluginhandler</code>	Required	The name in the Plugins registry for an implementation of the <code>InboundIntegrationHandlerPlugin</code> plugin interface. Note that this is the <code>.gwp</code> file name, not the implementation class name.	<code>InboundJMSIntegrationExample</code>
<code>transactional</code>	Required	You must always set this to <code>true</code> .	<code>true</code>
<code>threadpool</code>	Required	The unique name of a thread pool as configured earlier in the file. See “Thread Pool Configuration” on page 269.	<code>gw_default</code>

Plugin parameters in Plugins editor, description	Required	Description	Example value
ordered	Optional	<p>For typical use, set to <code>true</code> to maintain the processing of inbound JMS messages in order.</p> <p>The default value is <code>false</code>, which means the JMS integration dispatches the inbound messages in parallel with no guarantees of strict ordering.</p> <p>The behavior of the <code>ordered</code> flag with the polling and throttle interval works the same in the JMS integration as in the file integration. See “Understanding the Polling Interval and Throttle Interval” on page 285.</p>	<code>true</code>
batchlimit	Required	The maximum number of messages to receive in a poll interval	2
connectionfactoryjndi	Required	The application server configured JNDI connection factory	jms/gw/queueCF
destinationjndi	Required	The application server configured JNDI destination	jms/gw/queue
user	Optional	User name for authenticating on the JMS queue.	jsmith
password	Optional	Password for authenticating on the JMS queue.	pw123
osgiservice	Required	Always set to <code>true</code> . This is for internal use. This value is independent of whether you choose to register your handler class as an OSGi plugin.	<code>true</code>
stoponerror	Required	If <code>true</code> , PolicyCenter stops the integration if an error occurs. Otherwise, PolicyCenter just skips that message. Be sure to log any errors or notify an administrator.	<code>true</code>
messagereceivetimeout	Optional	The maximum time in seconds to wait for an individual JMS message. The default is 15.	15
pollinginterval	Optional	The time interval in seconds between polls, though the algorithm interacts with the throttle interval and the <code>ordered</code> parameter. See “Understanding the Polling Interval and Throttle Interval” on page 285. The default is 60 seconds.	60
throttleinterval	Optional	The time interval in seconds after polling, though the algorithm interacts with the polling interval and the <code>ordered</code> parameter. See “Understanding the Polling Interval and Throttle Interval” on page 285. The default is 60 seconds.	60

If you throw an exception in your code, the transaction of the message processing is rolled back. The original message is back in the queue.

5. In Studio, within the Plugins registry, add a new .gwp file. For related information, see “Registering a Plugin Implementation Class” on page 126 and “Using the Plugins Registry Editor” on page 131 in the *Configuration Guide*.
6. 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:

- For a message reply plugin, type `InboundIntegrationMessageReply`.
 - For a startable plugin for non-messaging use, type `InboundIntegrationStartablePlugin`.
7. Click the plus (+) symbol to add a plugin implementation and choose **Add Java plugin**.
8. In the **Java class** field, type:
- 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`.
9. 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.
10. Write your own *inbound integration handler plugin* implementation. Your handler code must implement a plugin interface
 - 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`.You can implement your plugin code in any method you choose: Gosu, Java (with no OSGi), or in OSGi (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.
This interface has one method called `process`, which has a single argument of type `Object`. The method returns no value. The JMS integration calls that method to process one message. Downcast this `Object` to `javax.jms.Message` before using it.
If you throw an exception in your code, the behavior depends on the configuration parameter `stopOnError`. If that parameter has the value `true`, processing on that queue stops. If it has the value `false`, that message is skipped. Be sure to catch any errors in your processing code and log any issues so that an administrator can follow up later.
11. In Studio, within the Plugins registry, register your handler plugin implementation class. Add a row in the Plugins registry editor for your plugin implementation class. See “Registering a Plugin Implementation Class” on page 126 and “Using the Plugins Registry Editor” on page 131 in the *Configuration Guide*. Set the interface name to the handler interface that you implemented (see step 10).
-
- IMPORTANT** Within the Plugins registry for your handler plugin implementation, the Name field must match the `pluginHandler` parameter you use in the `inbound-integration-config.xml` file for this integration.
-
12. 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.
13. Start the server and test your new inbound integration.

Custom Inbound Integrations

PolicyCenter includes built-in inbound integrations of file-based input and JMS messages. If these built-in integrations do not serve your needs, write your own integration based on the plugin interfaces in the `gw.plugin.integration.inbound` package:

- `InboundIntegrationMessageReply` – message reply plugin
- `InboundIntegrationStartablePlugin` – startable plugin for other non-messaging contexts

You can implement your plugin code in any method you choose: Gosu, Java (with no OSGi), or in OSGi (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.

The `InboundIntegrationStartablePlugin` plugin interface extends several other interfaces:

- `InitializablePlugin` – This interface requires one method that passes plugin parameters. The plugin parameters are passed to the `setup` method in the `WorkAgent` interface, which your plugin must implement.
- `WorkAgent` – This *work agent* interface defines the core behavior of the inbound integration framework. This is the most complex part of writing your own custom inbound integration. See “Writing a Work Agent Implementation” on page 279.
- `IStartablePlugin` – The startable plugin interface defines methods such as `start`, `stop`, and `getState`. See “Startable Plugins Overview” on page 259. Be aware that the main `WorkAgent` interface defines `start` and `stop` methods with no arguments. The `IStartablePlugin` interface adds additional method signatures of the `start` and `stop` methods that take arguments.

The `InboundIntegrationMessageReply` plugin interface extends several other interfaces:

- `MessageReply` – The interface for classes that handle replies from messages sent by a `MessageTransport` implementation. See “Implementing Messaging Plugins” on page 333 for the relationship between the interfaces `MessageTransport`, `MessageRequest`, and `MessageReply`.
- `WorkAgent` – This *work agent* interface defines the core behavior of the inbound integration framework. This is the most complex part of writing your own custom inbound integration. See “Writing a Work Agent Implementation” on page 279.

There are no other methods on `InboundIntegrationStartablePlugin` not defined in one of those other interfaces.

After you write your own custom implementation of `InboundIntegrationStartablePlugin`, use the PolicyCenter Studio Plugins Registry to register your plugin implementation with PolicyCenter. See “Using the Plugins Registry Editor” on page 131 in the *Configuration Guide*.

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. In addition to the primary functions of each method, you may also want to perform some logging to help debug your code.

To write a complete work agent implementation, you must write multiple related classes that work together. Refer to the following table for a summary of each class

Class that you write	Implements this interface	Description
A work agent	<code>WorkAgent</code>	Your plugin is the top level class that coordinates work for this service. Instantiates your class that implements the interface <code>Factory</code> .
Finding and preparing work during each polling interval		
A factory	<code>Factory</code>	A factory is a class that for each polling interval. Instantiates your class that implements the interface <code>WorkSetProcessor</code> .
A work set processor	<code>WorkSetProcessor</code> If your plugin supports the optional feature of being <i>transactional</i> , implement the subinterface <code>TransactionalWorkSetProcessor</code>	An object that knows how to acquire and divide resources. Instantiates your class that implements the interface <code>Inbound</code> . The main method for processing one unit of work within a <code>WorkData</code> object is the <code>process</code> method within this object.

Class that you write	Implements this interface	Description
Inbound	Inbound	A class that knows how to find work in its <code>findWork</code> method and return new work data sets. Instantiates your class that implements the interface <code>WorkDataSet</code> .
Representing the work itself		
A work data set	<code>WorkDataSet</code>	Represents the set of all data found in this polling interval. Encapsulates a set of work data (<code>WorkData</code>) objects and any necessary context information to operate on the data. Instantiates your class that implements the interface <code>WorkData</code> .
Work data	<code>WorkData</code>	Represents one unit of work

Setup and Teardown 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. The method signature is:

```
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.

Start and Stop the Plugin

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 different methods `setup` and `teardown`. See “Setup and Teardown the Work Agent” on page 280.

Handling Start and Stop in Custom Startable Plugins

If you use the startable plugin variant of in the inbound integration plugin (`InboundIntegrationStartablePlugin`), be aware that the main `WorkAgent` interface defines `start` and `stop` methods with no arguments.

The startable plugin variant also implements the interface `IStartablePlugin`. This means you must add additional method signatures of the `start` and `stop` methods that take arguments. See related topic “Startable Plugins Overview” on page 259.

For startable plugin variants of the custom inbound integration plugin, all method signatures of the `start` and `stop` method must call the appropriate method of the utility class `gw.api.integration.inbound.CustomWorkAgent`. As a method argument, pass the plugin name that you used in your `inbound-integration-config.xml` file with the `name` attribute on the `<custom-integration>` element.

- In each `start` method, call `CustomWorkAgent.startCustomWorkAgent(pluginName)`
- In each `stop` method, call `CustomWorkAgent.stopCustomWorkAgent(pluginName)`

The following Java example demonstrates this pattern:

```
import gw.api.integration.inbound.CustomWorkAgent;
import gw.api.server.Availability;
import gw.api.server.AvailabilityLevel;
import gw.api.startable.IStartablePlugin;

@Availability(AvailabilityLevel.MULTIUSER)
public class CustomStartableInboundIntegration implements IStartablePlugin {
    private String _name = "exampleCustomIntegration";
```

```
...
private void start() throws GWLifecycleException {
    CustomWorkAgent.startCustomWorkAgent( _name );
}

private void stop() {
    try {
        CustomWorkAgent.stopCustomWorkAgent( _name );
    }
    catch ( GWLifecycleException e ){
        throw new RuntimeException( e );
    }
}
```

Declare Whether Your Work Agent is Transactional

PolicyCenter calls the plugin method `transactional` to determine whether your agent is *transactional*. A transactional work agent creates work items with a slightly different interface. There are additional methods that you must implement to begin work, to commit work, and to roll back transactional changes to partially finished work. To specify your work agent is transactional, return `true` from the `transactional` method. Otherwise, return `false`.

If your work agent is transactional, your implementation of `Factory.createWorkUnit()` must return an instance of `TransactionalWorkSetProcessor` instead of `WorkSetProcessor`. The `TransactionalWorkSetProcessor` interface is a subinterface of (extends from) `WorkSetProcessor`. See “Get a Factory for the Work Agent” on page 281.

Get 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 has only one method, which is called `createWorkProcessor`. It 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 `Factory.createWorkProcessor()` must return an instance of `TransactionalWorkSetProcessor` instead of `WorkSetProcessor`. The `TransactionalWorkSetProcessor` interface is a subinterface of `WorkSetProcessor`. See “Declare Whether Your Work Agent is Transactional” on page 281. Both interfaces are in the `gw.api.integration.inbound` package.

Writing a Work Set Processor

Most of your actual work happens in code called a work set processor, which is a class that you create that implements the `WorkSetProcessor` interface or its subinterface `TransactionalWorkSetProcessor`. Both interfaces are in the `gw.api.integration.inbound` package. See “Declare Whether Your Work Agent is Transactional” on page 281

The basic `WorkSetProcessor` interface defines two methods

- `getInbound` – Gets an object that knows how to acquire and divide resources to create work items. This method returns an object of type `Inbound`, which is an interface with only one method, called `findWork`. Define your own class that implements the `Inbound` interface. The `findWork` method must get the *work data set*, which represent 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. See “Error Handling” on page 282. 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` is one work item in the work data set. You can optionally choose to use the `WorkContext` to declare a resource or other context necessary to process the data item. Your own implementation of the work data set (`WorkDataSet`) is responsible for populating 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. See “Error Handling” on page 282.

Only if your agent is transactional, your implementation of `Factory.createWorkUnit()` must return an instance of `TransactionalWorkSetProcessor` instead of `WorkSetProcessor`. The `TransactionalWorkSetProcessor` interface is a subinterface of `WorkSetProcessor`. See “Declare Whether Your Work Agent is Transactional” on page 281.

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 is a subinterface of `WorkContext`. Use it to represent customer-specific work context information that also contains transaction-specific information. Create your own implementation of this class in your `getContext` method of your `WorkDataSet`. See “Creating a Work Data Set” on page 282.

Error Handling

Any exceptions in `WorkDataSet.findWork()` causes PolicyCenter to immediately stop processing until the plugin is restarted or the server is restarted.

Any exceptions in the `WorkSetProcessor` in the `process` method are logged and the item is skipped.

WARNING It is important to catch any exceptions in the `process` method to ensure that you correctly handle error conditions. For example, you may need to notify administrators or place the work item in a special location for special handling.

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 own 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. See “Get a Factory for the Work Agent” on page 281.

Create a class that implements the `gw.api.integration.inbound.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.

Note: From Gosu, 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 code that processes each item (your `WorkSetProcessor` implementation), you can access this connection information from the `WorkSetProcessor` object's `process` method when processing each new message. If your plugin supports transactional work items (the `transactional` plugin parameter), your class must implement a subinterface called `TxContext`, which requires two additional methods:
 - `isRollback` – Returns a `boolean` value that indicates the transaction will be rolled back.
 - `setRollbackOnly` – Set your own `boolean` value to indicate that a rollback will occur.
- `close` – Close and release any resources acquired by your work data set.

Getting Parameters

Like all other plugin types, your plugin implementation can get parameter values. See “Plugin Parameters” on page 127. 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. See the parameter called `parameters` in “Installing a New Custom Inbound Integration” on page 283.

Installing a New Custom Inbound Integration

To create and register a new custom inbound integration

1. Design and write all required implementation Java classes as described in “Installing a New Custom Inbound Integration” on page 283.

IMPORTANT You must write your plugin implementation for this plugin interface in Java, not Gosu.

2. In the Project window, navigate to `configuration` → `config` → `integration`, and the open `inbound-integration-config.xml`.
3. Configure the thread pools. See “Thread Pool Configuration” on page 269.
4. In the list of integrations, create one `<integration>` element of type `<custom-integration>`. Follow the pattern in the file to set the XML element namespace. In the base configuration, type the element name as `<ci:>custom-integration`. Set the `name` and `disabled` attributes as described in “Configuring a List of Inbound Integrations” on page 270.

5. Set configuration parameter subelements as follows:

Plugin parameters in Plugins editor, description	Required	Description	Example
<code>workagentimpl</code>	Required	The name of the <code>InboundIntegrationStartablePlugin</code> or <code>InboundIntegrationMessageReply</code> implementation class. If this is a Java class, list the fully-qualified name of the class with the package. If this is a Gosu class, list the fully-qualified name of the class with the package, with the suffix <code>.gs</code> . For example, <code>mycompany.integ.MyInboundPlugin.gs</code>	<code>mycompany.integ.MyInboundPlugin.gs</code>
<code>pluginhandler</code>	n/a	<i>This parameter is unused in custom inbound integrations. This parameter is used only for file and JMS integrations.</i>	n/a
<code>transactional</code>	Optional	Always set this boolean value to the same value returned by your plugin implementation's <code>transactional</code> method. See "Declare Whether Your Work Agent is Transactional" on page 281. If not set, defaults to <code>false</code> .	true
<code>threadpool</code>	Optional	The unique name of a thread pool as configured earlier in the file. See "Thread Pool Configuration" on page 269.	<code>gw_default</code>
<code>stoponerror</code>	Required	If true, PolicyCenter stops the integration if an error occurs. Otherwise, PolicyCenter just skips that item. Be sure to log any errors or notify an administrator.	true
<code>osgiservice</code>	Required	If you deploy your plugin implementation as an OSGi plugin, set this to <code>true</code> . If you deploy your plugin implementation as a Gosu plugin, set this to <code>true</code> . If you deploy your plugin implementation as a Java plugin (not using OSGi), set this to <code>false</code> .	true
<code>ordered</code>	Optional	Set to <code>true</code> to process in a single thread. Set to <code>false</code> to process items in multiple threads, as managed by the thread pool. The behavior of the <code>ordered</code> flag interacts with the behavior of the polling interval and throttle interval. See "Understanding the Polling Interval and Throttle Interval" on page 285.	true
<code>pollinginterval</code>	Optional	The time interval in seconds between polls, though the algorithm interacts with the throttle interval and the <code>ordered</code> parameter. See "Understanding the Polling Interval and Throttle Interval" on page 285. The default is 60 seconds.	60

Plugin parameters in Plugins editor, description	Required	Description	Example
throttleinterval	Optional	The time interval in seconds after polling, though the algorithm interacts with the polling interval and the ordered parameter. See “Understanding the Polling Interval and Throttle Interval” on page 285. The default is 60 seconds.	60
parameters	Optional	Arbitrary parameters that you can define, for example to store server names and port numbers. Define subelements that alternate between key and value elements, contain key/value pairs. For example: The plugin interface has a setup method. These parameters are in the java.util.Map that PolicyCenter passes to that initialization method.	<parameters> <key>key1</key> <value>myvalue</value> </parameters>

6. In Studio, within the Plugins registry, add a new .gwp file. For related information, see “Registering a Plugin Implementation Class” on page 126 and “Using the Plugins Registry Editor” on page 131 in the *Configuration Guide*.
7. Studio prompts for a plugin name and plugin interface. For the name, use a name that represents the purpose of this specific inbound integration.
8. For the **Interface** field, type the plugin interface you implemented, either `InboundIntegrationStartablePlugin` or `InboundIntegrationMessageReply`.
9. Click the plus (+) symbol to add a plugin implementation and choose **Add Java plugin**.
10. In the **Java class** field, type the fully-qualified name of your plugin implementation.
11. 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.
12. Start the server and test your new inbound integration. Add logging code as appropriate to confirm the integration.

Understanding the Polling Interval and Throttle Interval

There are two different configuration parameters for coordinating when the work begins: `pollinginterval` and `throttleinterval`.

The primary mechanism is the *polling interval* (`pollinginterval`). Additionally, there is a throttle interval (`throttleinterval`) which you can use to reduce the impact on the server load or external resources.

1. At the beginning of the polling interval, the integration polls for new work and creates new work items but not yet begin the items.
2. PolicyCenter begins working on the work items.
 - If the work is ordered (the `ordered` parameter is `true`), the work happens in the same thread.
 - If the work is unordered (the `ordered` parameter is `false`), the work happens in separate additional threads with no guarantee of strict ordering.
3. After the current work is complete, the system determines how much time has transpired and compares with the two interval parameters. The behavior is slightly different for ordered and unordered work.

If the work is ordered (the `ordered` parameter is `true`), the following table describes the behavior.

If the total time since last polling is...	Behavior
Less than the <code>pollinginterval</code>	The server waits until the remaining part of the polling interval and then waits the complete <code>throttleinterval</code> time
Greater than the <code>pollinginterval</code> but less than the sum of the <code>pollinginterval</code> and the <code>throttleinterval</code>	The server waits until the end of the sum of the <code>pollinginterval</code> and the <code>throttleinterval</code> times
Greater than the sum of the <code>pollinginterval</code> and the <code>throttleinterval</code>	The server immediately polls for new work

If the work is unordered, the same time check occurs with some differences:

- the work proceeds in parallel
- the time check happens immediately after new work is created but does not wait for the work to be done.

part IV

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. Write rules to generate messages in response to events of interest. PolicyCenter queues these messages and then dispatches them to the receiving systems.

This topic explains how PolicyCenter generates messages in response to events and how to connect external systems to receive those messages.

This topic discusses *plugins*, which are software modules that PolicyCenter calls to perform an action or calculate a result. For information on plugins, see “Plugin Overview” on page 123. For the complete list of all PolicyCenter plugins, see “Summary of All PolicyCenter Plugins” on page 141.

Register new messaging plugins in Studio. As you register plugins in Studio, Studio prompts you for a plugin interface name (in a picker) and, in some cases, for a plugin name. Use that plugin *name* as you configure the messaging destination in the Messaging editor in Studio.

This topic includes:

- “Messaging Overview” on page 290
- “Message Destination Overview” on page 302
- “Filtering Events” on page 309
- “List of Messaging Events in PolicyCenter” on page 309
- “Generating New Messages in Event Fired Rules” on page 317
- “Message Ordering and Multi-Threaded Sending” on page 323
- “Late Binding Data in Your Payload” on page 328
- “Reporting Acknowledgements and Errors” on page 329
- “Tracking a Specific Entity With a Message” on page 333
- “Implementing Messaging Plugins” on page 333
- “Resynchronizing Messages for a Primary Object” on page 339

- “Message Payload Mapping Utility for Java Plugins” on page 342
- “Message Status Code Reference” on page 343
- “Monitoring Messages and Handling Errors” on page 344
- “Messaging Tools Web Service” on page 347
- “Batch Mode Integration” on page 348
- “Included Messaging Transports” on page 349

Messaging Overview

To understand this topic, it might help to get a high-level overview of terminology and an overview of events in PolicyCenter. The following subtopics summarizes important messaging concepts.

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. For some (but not all) entities in the data model configuration files, if a user action or API adds, changes, or removes an entity instance, the system triggers an event. Each event has a name, which is a `String` value.

For example, in PolicyCenter a new policy submission triggers an event. The event name is `"IssueSubmitted"`.

One user action or API call might trigger multiple events for different objects in one database transaction. In some cases, the object might have multiple events occur in one database transaction.

Triggering an event triggers one call to the Event Fired rule set for each external system that is interested in that event.

Event Fired rules for an event name run only if you tell PolicyCenter that at least one external system is interested in that event. To request that PolicyCenter run Event Fired rules for that event and that destination ID, see “Message Destination Overview” on page 302.

Additionally, only entity types defined with the `<events>` tag by default generate added, changed, or deleted events. For more information, see “`<events>`” on page 210 in the *Configuration Guide*.

Message

A message is information to send to an external system in response to an event. PolicyCenter can ignore the event or send one or more messages to each external system that cares about that event. In addition to an ID and some status information, each message has a *message payload*. The payload is the main data content of the message. The payload is `String` in the `Message.Payload` property.

Your code creates messages in the Event Fired rule set. For more information, see “Generating New Messages in Event Fired Rules” on page 317. Because message creation impacts user response times, avoid unnecessarily large or complex messages if possible.

Message History

After a message is sent, the application converts a message (`Message`) object to a message history (`MessageHistory`) object. The message history object has the same properties as a message object.

You can use the message history database table to understand the messaging history to your external systems. This can be useful to help understand problems.

Additionally, message history objects are important when detecting duplicate messages. To report a duplicate message, you must find the message history object. Call the `reportDuplicate` method on the message history object, which represents the original message. See “[Reporting Acknowledgements and Errors](#)” on page 329.

Messaging Destinations

A messaging destination is an external system to which to send messages. Generally speaking, a messaging destination represents one external system. Register one destination for each external system, even if that external system is used for multiple types of data.

Register your messaging destinations in Studio. In the **Messaging** editor, specify which classes implement the messaging plugin interfaces for that messaging destination. The most important message plugin interface is `MessageTransport`. The `MessageTransport` plugin interface is responsible for actually dispatching the message (most importantly, its payload) to your external system.

Each destination registers a *list of event names* for which it wants notifications. Each destination may listen for different events compared to other destinations. The Event Fired rule set runs once for every combination of an event name and a destination interested in that event. To request that PolicyCenter run Event Fired rules for that event and that destination ID, see “[Message Destination Overview](#)” on page 302.

If more than one messaging destination requests an event name, PolicyCenter runs the Event Fired rule set once for every destination that requested it. To determine which destination this event trigger is for, your Event Fired rules must check the `DestinationID` property of the message context object at run time.

In the **Messaging** editor there are other important settings for each destination, including:

- **Alternative Primary Entity** – See “[Primary Entity and Primary Object](#)” on page 291
- **Message Without Primary** – See “[How Destination Settings Affects Ordering](#)” on page 327

Root Object

A root object for an event is the entity instance most associated with the event. This might be a small subobject or a more prominent high-level object.

A root object for a message is the entity instance most associated with the message. This might be a small subobject or a more prominent high-level object.

By default, the message’s root object is the same as the root object for the event that created the message in your Event Fired rules. This default makes sense in most cases. You can override this default for a message if desired. See “[Setting a Message Root Object or Primary Object](#)” on page 321.

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.

IMPORTANT Determining which primary object, if any, applies for a messaging destination is critical to understanding how PolicyCenter orders messages. See “[Safe Ordering](#)” on page 292 and “[Message Ordering and Multi-Threaded Sending](#)” on page 323.

In PolicyCenter:

- 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.

- A messaging destination can specify the **Contact** entity as an alternative primary object, in which case that setting applies just to that messaging destination.
- No other entity types can be alternative primary entities for a messaging destination.

Some objects are not associated with any primary object. For example, **User** objects are not associated with a single policy. Messages associated with such objects are called *non-safe-ordered messages*. See “Safe Ordering” on page 292 and “Message Ordering and Multi-Threaded Sending” on page 323.

Do not confuse the *root object* for a message with the *primary object* associated with a message. The root object is the generally 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. See “Setting a Message Root Object or Primary Object” on page 321.

Acknowledgement (ACK and NAK)

An acknowledgement is a formal response from an external system back to a PolicyCenter that declares how successfully the system processed the message:

- A *positive acknowledgement* (ACK) means that the external system processed the message successfully.
- A *negative acknowledgement* (NAK) means the external system failed to handle the message due to errors.

It is very important for integration programmers to understand that PolicyCenter distinguishes between the following errors:

- An error that throws a Gosu or Java exception during initial sending in the `MessageTransport` method called `send`. Errors during this phase typically indicate network errors or other automatically retryable errors. If the `send` method throws an exception, PolicyCenter automatically retries sending the message multiple times.
- An error reported from the external system, which are also called a NAK.

For more information about error handling, see “Error Handling in Messaging Plugins” on page 338.

Safe Ordering

Safe ordering is a messaging feature that causes related messages to be received in the same order they were sent. This is called *safe order*. Messages are grouped by their related *primary object* for each messaging destination. For important definitions relevant to this topic, see “Primary Entity and Primary Object” on page 291.

Messages send in creation order with other messages associated with that same primary object for that destination. Any messages associated with a primary object for that destination are called *safe-ordered messages*. The application waits for an acknowledgement before processing the next safe-ordered message for that same primary object for that destination. In other words, delays or errors for that destination block further sending of messages for that same destination for that same primary object.

In PolicyCenter, by default PolicyCenter sends any messages associated with an account grouped by account for each messaging destination and sent in creation order. These are *safe-ordered messages*. If a messaging destination sets **Contact** as its alternative *primary entity*, messages associated with a contact send grouped by contact for each messaging destination and sent in creation order.

There are PolicyCenter objects that are not associated with a primary entity. For example, **User**. All messages that are not associated with a claim (the primary object) are called *non-safe-ordered messages*.

Not all messages are associated with a primary object. The logic for how and when to send these *non-safe-ordered messages* is different from the logic for safe-ordered messages. Additionally, the behavior varies based on the messaging destination configuration setting called **Strict Mode**. For details, see “Message Ordering and Multi-Threaded Sending” on page 323.

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:

- **Submit the message by using remote API calls** – Use a web service (SOAP) interface or a Java-specific interface to send a message to an external system.
- **Submit the message to a guaranteed-delivery messaging system** – For instance, 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.

Messaging Flow Overview

The high-level steps of event and message generation and processing are as follows.

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 messaging 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 messaging event represents the changes to application data as the entity commits to the database.

3. Event Fired rules create messages – PolicyCenter runs the Event Fired rule set for each event name that triggers. PolicyCenter runs the rule set multiple times for an event name if multiple destinations listen for it. Your rules can choose to generate new messages. Messages have a text-based *message payload*.

In PolicyCenter, for example you might write rules that check if the event name is `IssueSubmission`, and if certain other conditions occur. If so, the rules might generate a new message with an XML payload. The rules might also link entities to the message for later use.

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 acknowledgement – The external system replies with an acknowledgement to the destination after it processes the message, and the destination's messaging plugins process this information. If the message successfully sent, the messaging plugins submit an ACK and PolicyCenter sends the next message. For details of the messaging ordering system, see “Message Ordering and Multi-Threaded Sending” on page 323. The code that submits the acknowledgement might also make changes to application data, such as updating a messaging-specific property on a persisted object or advancing a workflow. If you change data, see the warnings in the section “Messaging Flow Details” on page 295. Remember that you can also submit an error (a negative acknowledgement, a NAK) instead of a positive acknowledgement.

Most messaging integration code that you write for a typical deployment is writing your Event Fired rules that create message payloads and writing `MessageTransport` plugin implementations to send messages. There are two other optional types of messaging plugins and discussed later.

Once you write messaging plugin implementation classes, you register your new messaging plugin classes in Studio. Register your messaging plugin implementations first in the **Plugins** editor and then in the **Messaging** editor. For plugin interfaces that can have multiple implementations, which includes all messaging plugin interfaces, Studio asks you to *name* your plugin implementation when registering your plugin class. Use that *plugin implementation class name* when you configure the messaging destination in the **Messaging** editor.

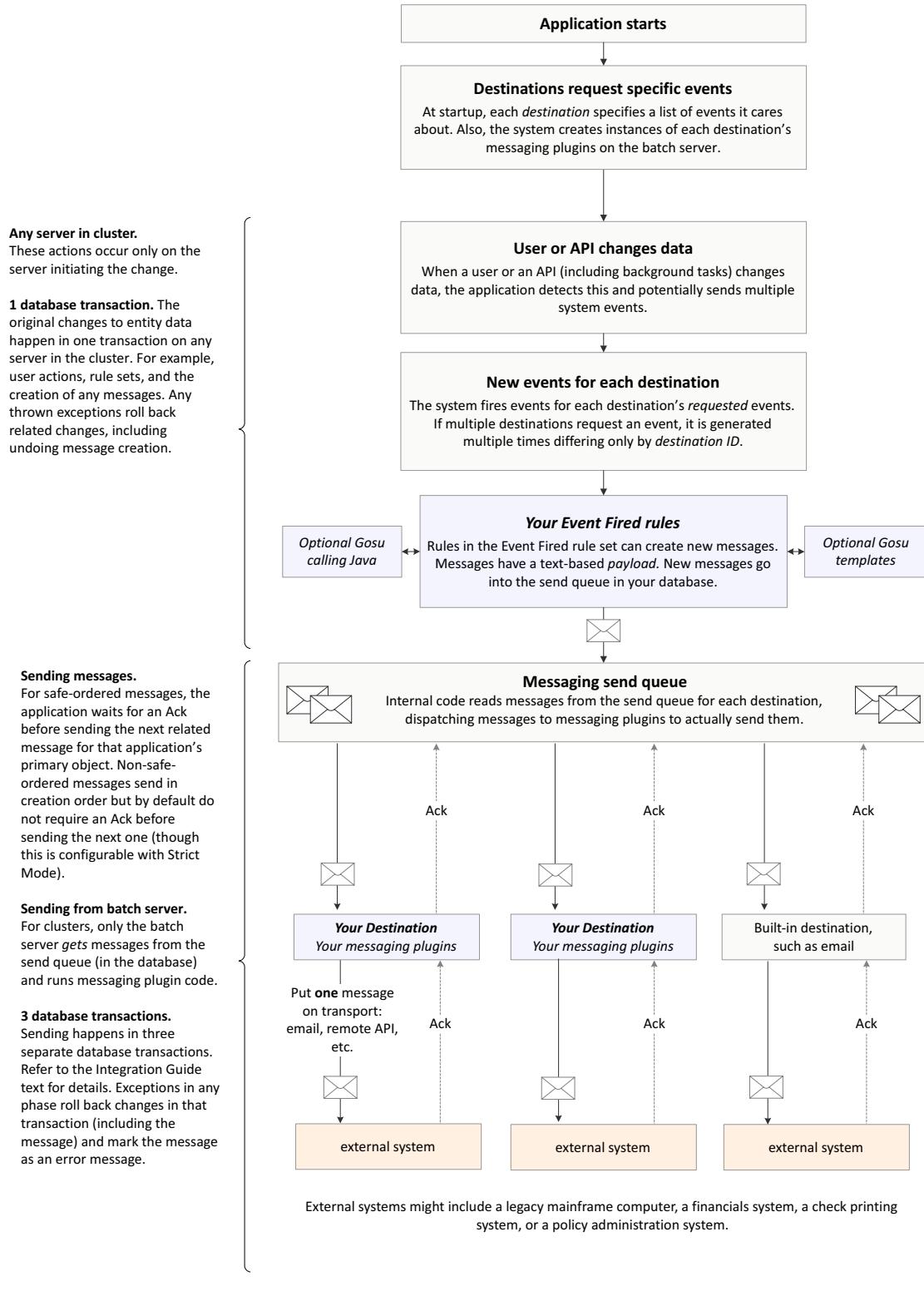
See Also

- “Using the Messaging Editor” on page 153 in the *Configuration Guide*
- “Using the Plugins Registry Editor” on page 131 in the *Configuration Guide*
- “Plugin Overview” on page 123
- “Message Destination Overview” on page 302

Messaging Flow Details

The following diagram illustrates the chronological flow of events and messaging, starting from the top of the diagram. Refer to the section following the diagram for a detailed explanation of each step.

Messaging Overview



The following list describes 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. PolicyCenter saves a list of events for which each destination requested notifications. Because this happens at system startup, if you change the list of events or destination, you must restart PolicyCenter. 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, and the message reply plugin handles message replies.

Register new messaging plugins in Studio first in the Plugins editor. When you create a new implementation, Studio prompts you for a plugin interface name (in a picker) and, in some cases, for a plugin name. Use that plugin name in the Messaging editor in Studio to register each destination. Remember that you need to register your plugin in two different editors in Studio.

2. **A user or an API changes something** – A user action, API, or a batch process changed data in PolicyCenter.

For example, PolicyCenter triggers an event if you add a policy or change a note.

It is critical to understand that the change does not fully commit to the database until step 4. Any exceptions that occur before step 4 undo the change that triggered the event. In other words, unless all follow-up actions to the change succeed, the database transaction rolls back. For related information, see “Messaging Database Transactions During Sending” on page 308.

PolicyCenter generates messaging events – The same action, such as a single change to the PolicyCenter database, 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 the same except with different *destination ID* properties. It is critical to understand that a change to PolicyCenter data might generate events for one destination but not for another destination. To change this list for destination, review the **Messaging** editor in Studio and select the row for your destination. Additionally, only entity types defined with the `<events>` tag by default generate added, changed, or deleted events. For more information, see “`<events>`” on page 210 in the *Configuration Guide*.

3. **PolicyCenter invokes Event Fired rules (and they generate messages)** – PolicyCenter calls the Event Fired rule set for each destination-event pair. Event Fired rules must rules 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 using the Gosu method `messageContext.createMessage(payload)`. The rule actions choose whether to generate a message, the order of multiple messages, and the text-based message *payload*. For more information, see “Generating New Messages in Event Fired Rules” on page 317. Your rules can use the following techniques:

- **Optionally export an entity to XML using generated XSDs** – 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. For more details, see “Creating XML Payloads Using Guidewire XML (GX) Models” on page 322.

The Guidewire XML (GX) modeler is a powerful tool for integrations. Create custom XML models that contain only the subset of entity object data that is appropriate for each integration point. It can output a custom XSD that your external system can use. You can also use GX models to parse (import) XML data into in-memory objects that describe the XML structure. See “Creating XML Payloads Using Guidewire XML (GX) Models” on page 322.

- **Optionally use Gosu templates to generate the payload** – Rules can optionally use templates with embedded Gosu to generate message content.

- **Optionally use Java classes (called from Gosu) to generate the payload** – Rules can optionally use Java classes to generate the message content from Gosu business rules. See “Calling Java from Gosu” on page 123 in the *Gosu Reference Guide*.
- **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. See “Late Binding Data in Your Payload” on page 328 for details.

WARNING Event Fired rules and messaging plugin implementations have limitations about what data you can change. See “Restrictions on Entity Data in Messaging Rules and Messaging Plugins” on page 299.

4. New messages are added to the send queue – After all rules run, PolicyCenter adds any new messages to the send queue in the database. The submission of messages to the send queue is part of the same database transaction that triggered the event to preserve atomicity. If the transaction succeeds, all related messages successfully enter the send queue. If the transaction *fails*, the change rolls back including all messages added during that transaction. Messages might wait in this queue for a while, depending on the state of acknowledgements and the status of safe ordering of other messages (see step 5).

5. On the batch server only, PolicyCenter dispatches messages to messaging destinations – PolicyCenter retrieves messages from the send queue and dispatches messages to destinations. The messaging plugins for each destination sends each message. For details of how PolicyCenter retrieves messages and orders them for sending, see “Safe Ordering” on page 292 and “Message Ordering and Multi-Threaded Sending” on page 323.

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. See “Transport Neutrality” on page 293. If the `send` method throws an exception, PolicyCenter automatically retries the message.

6. Acknowledging messages – Some destination implementations know 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 that message.

In either case, the messaging destination code or the external system must confirm that the message arrived safely by submitting an *acknowledgement* (an ACK) to PolicyCenter for that specific message. Alternatively, it can submit an error, also called a negative acknowledgement or 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 acknowledgement or error.

If using messaging plugins to submit the ACK, you can also make changes to data during the ACK, such as updating properties on entities. Or, where appropriate, you could advance a workflow (see the following section).

Event Fired rules and messaging plugin implementations have limitations about what data you can change. See “Restrictions on Entity Data in Messaging Rules and Messaging Plugins” on page 299.

7. 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. For details of how messages are retrieved and ordered, see “Safe Ordering” on page 292 and “Message Ordering and Multi-Threaded Sending” on page 323. Read that section for details of the setting called Strict Mode. Strict Mode affects whether PolicyCenter waits for acknowledgements for non-safe-ordered messages.

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.

WARNING For data integrity and server reliability, you must carefully follow restrictions regarding what data you can change in Event Fired rules and messaging plugin implementations.

Event Fired Rule Set Restrictions for Entity Data Changes

WARNING Entity changes in Event Fired must be very limited. Changes do not trigger validation rules, or pre-update rules. Read this topic carefully for additional restrictions.

The following important restrictions apply to code within the Event Fired rule set:

- In general, perform any data updates in your preupdate rules, not your 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 safe to add is a new message using the `createMessage` method on the message context object. Never create new objects of any other types, 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 these common rule sets:
 - Pre-update rule set
 - Validation rule set
 - Event Fired rule set
- These restrictions apply to all entity types, including custom entity types.

Design your messaging rules carefully around these restrictions to avoid data corruption and logical errors that are difficult to diagnose.

Messaging Plugin Restrictions for Entity Data Changes

WARNING Entity changes in messaging plugin code must be very limited. Changes do not trigger validation rules, pre-update rules, or concurrent data change exceptions. Read this topic carefully for additional restrictions.

The following important restrictions apply to code triggered by a `MessageTransport` or `MessageReply` plugin implementation:

- 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.
- You cannot create any new objects in general. In the default configuration, only the following objects are safe to create within a messaging plugin:
 - activities
 - notes
 - workflows and workflow items
 - work queues

You cannot rely on preupdate 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. For advice on specific changes, contact Guidewire Customer Support.

- Never delete objects.
 - You must not rely on any entity data changes eventually triggering these common rule sets:
 - Pre-update rule set
 - Validation rule set
 - Event Fired rule set for standard events `ENTITYAdded`, `ENTITYChanged`, and `ENTITYRemoved`.
- However, Event Fired rule set still triggers for events that you explicitly add. You can use the API `entity.addEvent("messageName")` to add events. PolicyCenter calls the Event Fired rule set once for each custom event for each messaging destination that listens for it. Your Event Fired code must conform to all restrictions in “Event Fired Rule Set Restrictions for Entity Data Changes” on page 299.
- 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. See later in this topic for additional guidance.
 - 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 long before messaging code runs. As mentioned earlier, it is unsupported to create a 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.

In a `MessageRequest` plugin implementation, do no data changes at all.

Separate from the concurrent data exception differences mentioned earlier, there is an optional feature to *lock* related objects during messaging actions. If you use optional locking and code tries to access locked data (the primary entity instance or the message), the calling code waits for it to be unlocked.

IMPORTANT For maximum data integrity, enable optional entity locking during messaging. See “Messaging Database Transactions During Sending” on page 308. For maximum performance, disable optional entity locking during messaging.

Design your messaging code carefully around these restrictions to avoid data corruption and logical errors that are difficult to diagnose.

Messaging Interacts With PolicyCenter Workflow

If you submit Acks with messaging plugins rather than using the 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 `Submission.finishIssue(PolicyPeriod)`. You also can call trigger invocation methods directly on the workflow objects, such as `workflow.invokeTrigger(WorkflowTriggerKey)`. 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 those files. 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 specific workflow trigger.

Database Transactions When Creating Messages

All steps up to and including adding messages to the send queue occur in one database transaction. In the list earlier in “Messaging Flow Details” on page 295, this is step 2 through step 4. All changes commit in the same transaction. This is always the same database transaction that triggers the initial event.

The database transaction rolls back if any of the following occur:

- 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

If any of these errors occur, PolicyCenter rolls back all messages added to the send queue in that transaction.

There are special rules about database transactions during message sending at the destination level. See “Messaging Database Transactions During Sending” on page 308.

Messaging in PolicyCenter Clusters

If you run PolicyCenter in a cluster, be aware that step 1 through step 4 can occur on any PolicyCenter server within the cluster. PolicyCenter processes events on the same server as the user action or API call that triggered the event.

Once a message is in the send queue (step 5 through step 7), any further action with the message occurs *only* on the batch server.

Consequently, the batch server is the only server on which messaging plugins run. Configure your batch server (and any backup batch servers) to communicate with your destination external systems. For example, remember to configure your firewalls accordingly including your backup batch servers.

For PolicyCenter clusters, only the batch server actually uses your messaging plugins.

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. 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. If you have questions about how to convert some particular use, locally-hosted SOAP APIs from plugins or rules, contact Customer Support.

This is true for all types of local loopback 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.

WARNING 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.

Message Destination Overview

To represent each external system that receives a message, you must define a *messaging 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.

Carefully choose how to structure your messaging code. For example, if there are several related remote systems or APIs, you must 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. This means that the definition of a destination is important for message ordering and multithreading. See “Message Ordering and Multi-Threaded Sending” on page 323.

Each destination specifies a list of events for which it wants notifications and various other configuration information. Additionally, a destination encapsulates a list of your plugins that perform its main destination functions:

- **Message preparation** – (Optional) If you need message preparation before sending, write a plugin that implements the message request (`MessageRequest`) plugin interface. For example, this plugin might take simple name/value pairs stored in the message payload and construct an XML message in the format required by a transport or final message recipient. If the destination requires no special message preparation, omit the request plugin entirely for the destination. For implementation details and optional post-send processing, see “Implementing a Message Request Plugin” on page 334.
- **Message transport** – (Required) The main task of a destination is to send messages to an external system. Its underlying protocol might be an external message queue, web service request to external systems, FTP, or a proprietary legacy protocol. You actually send your messages in your own implementation of the `MessageTransport` plugin interface. Every destination must provide a message transport plugin implementation. Multiple messaging destinations might use the same messaging transport plugin implementation if they really do use the underlying transport code. For implementation details, see “Implementing a Message Transport Plugin” on page 334.
- **Message reply handling** – (Required only if asynchronous) If a destination requires an asynchronous callback for acknowledgements, implement the `MessageReply` plugin. If the destination requires no asynchronous acknowledgement, omit the reply plugin. For implementation details, see “Implementing a Message Reply Plugin” on page 336.

Note: PolicyCenter includes multi-threaded 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`. See “Multi-threaded Inbound Integration Overview” on page 267 and “Custom Inbound Integrations” on page 278.

After you write code that implements your messaging plugins, register them in Studio. When registering an implementation of new messaging plugin, Studio prompts you for a plugin name. The plugin name is different from the implementation class name. The plugin name is a short arbitrary name that identifies a plugin implementation. Studio only prompts you for a plugin name for plugin interfaces that support more than one implementation.

Use that plugin name to configure the messaging destination in the Messaging editor in Studio to register each destination. For details and examples of this editor, see “Using the Messaging Editor” on page 153 in the *Configuration Guide*.

For more information about plugins, see “Plugin Overview” on page 123.

To create new destinations, configure the messaging registry to specify the list of destinations, each of which includes the following information:

- The *plugin name* class for a `MessageTransport` plugin in Java or Gosu.
- Optionally, the *plugin name* for the implementation of a `MessageRequest` plugin.
- Optionally, the *plugin name* for the implementation of a `MessageReply` plugin.
- **Retry Interval** – The amount of time in milliseconds (`initialretryinterval`) after a retryable error to retry a sending a message.
- **Max Retries** – The number of automatic retries (`maxretries`) to attempt before suspending the messaging destination.
- **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 5 minutes, and the multiplier (`retrybackoffmultiplier`) is set to 2, PolicyCenter attempts the next retry in 10 minutes.
- **Event Names** – A list of events to listen for, by name. Each event triggers the Event Fired rule set for that destination. To specify that the destination wants to listen for all events, use the special event name string “`(\w)*`”.
- **Destination ID**, which typically you use in your 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. This method allows your destination to get its own destination ID to store it in a private variable. Your code can use the stored value for logging messages or send it to integration systems so that they can programmatically suspend/resume the destination if necessary.

The valid range for your destination IDs is 0 through 63, inclusive. Guidewire reserves all other destination IDs for built-in destinations such as the email transport destination.

- **Alternative Primary Entity** – See “Primary Entity and Primary Object” on page 291
- **Strict Mode** – See “How Destination Settings Affects Ordering” on page 327
- **Poll interval** – 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 this amount of time passes. Use this field to set the value of the polling interval to wait. 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. See “Message Ordering and Multi-Threaded Sending” on page 323 for details on how the polling interval works. If your performance issues primarily relate to many messages per primary object per destination, then the polling interval is the most important messaging performance setting.
- **Sender Threads** – To send messages associated with a primary object, PolicyCenter can create multiple sender threads for each messaging destination to distribute the workload. These are threads that actually call the messaging plugins to send the messages. Use this field to configure the number of sender threads for safe-ordered messages. This setting is ignored for non-safe-ordered messages, since those are always handled by one thread for each destination. If your performance issues primarily relate to many messages but few messages per claim for each destination, then this is the most important messaging performance setting. For more information, see “Message Ordering and Multi-Threaded Sending” on page 323.
- **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. For more information about suspend and shutdown actions, see “Message Destination Overview” on page 302.

Manage these settings in Guidewire Studio in the Messaging editor. See “Using the Messaging Editor” on page 153 in the *Configuration Guide*.

IMPORTANT If you register a messaging plugin, you must register it in two places. First, register it in the plugin registry in the plugin editor; see “Using the Plugins Registry Editor” on page 131 in the *Configuration Guide*. Remember the plugin implementation name that you use. You need it to configure the messaging destination in the Messaging editor in Studio to register each destination. Next, register it in the messaging registry in the messaging editor; see “Using the Messaging Editor” on page 153 in the *Configuration Guide*.

To write your messaging plugins, see “Implementing Messaging Plugins” on page 333.

WARNING Event Fired rules and messaging plugin implementations have limitations about what data you can change. See “Restrictions on Entity Data in Messaging Rules and Messaging Plugins” on page 299.

Sharing Plugin Classes Across Multiple Destinations

It is common to implement messaging plugin classes that multiple destinations share. For example, a transport plugin might manage the transport layer for multiple destinations that use that physical protocol. In such cases, be aware of the following things:

- 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. For more information, see “Saving the Destination ID for Logging or Errors” on page 339.

Handling Acknowledgements

Due to differences in external systems and transports, there are two basic approaches for handling replies. PolicyCenter supports synchronous and asynchronous acknowledgements, although in different ways:

1. **Synchronous acknowledgement 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 acknowledgement approach. The synchronous approach requires that your transport plugin `send` method actually send the message and immediately submit the acknowledgement with the message method `reportAck`. For error and duplicate handling:
 - In most cases, 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. See “Submitting Acks, Errors, and Duplicates from Messaging Plugins” on page 330.
2. **Asynchronous acknowledgement** – 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 acknowledgement 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 acknowledgements, as described later.

For asynchronous acknowledgement, 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 acknowledgement, the destination reports to PolicyCenter that the message successfully sent.

One special step in asynchronous acknowledgement 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 (the ACK itself and or additional property updates) to the PolicyCenter database.

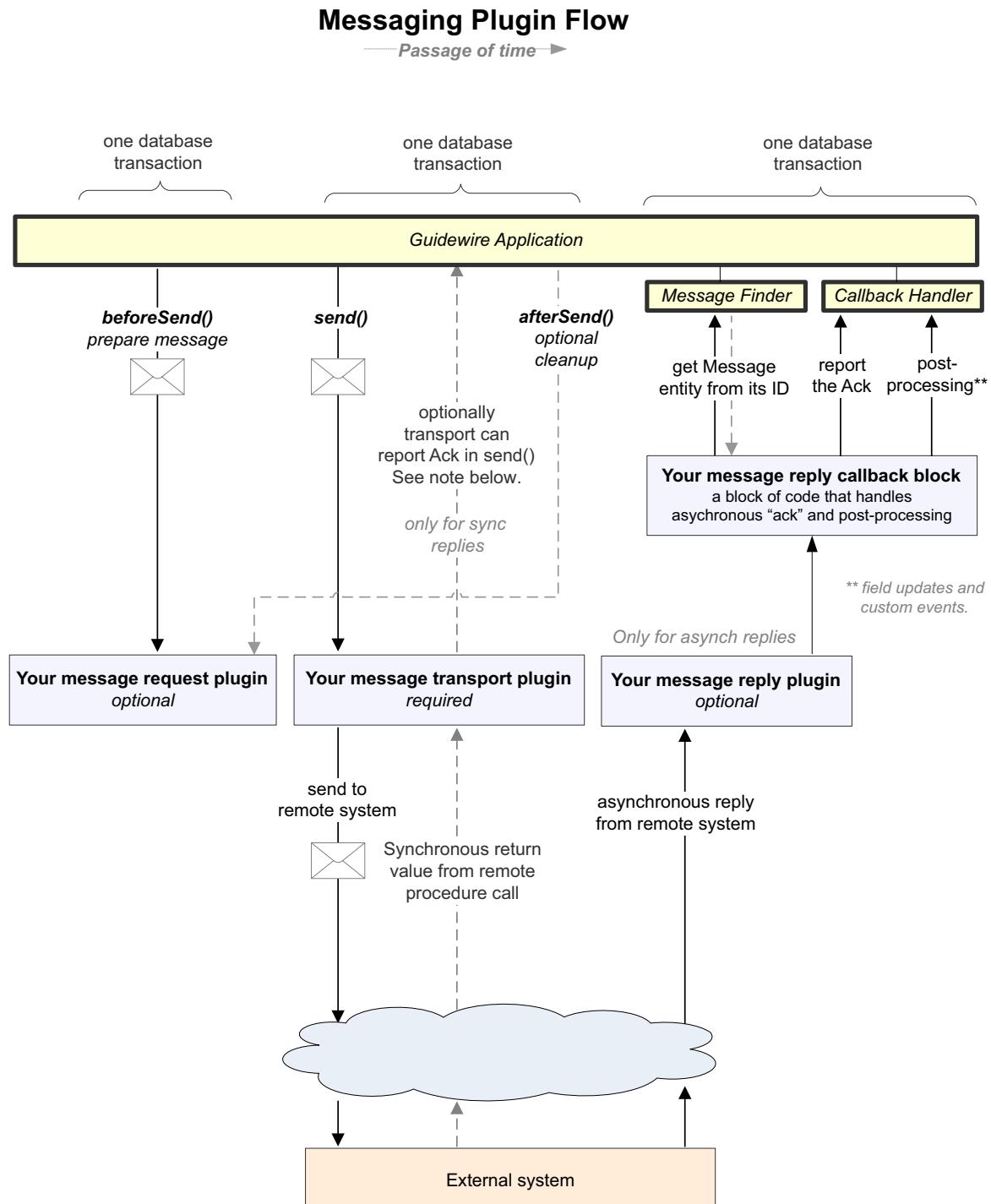
To set up the callbacks properly, Guidewire provides several interfaces, including:

- **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.

For more information about these objects and how to implement them, see “[Implementing a Message Reply Plugin](#)” on page 336.

An alternative to this approach is for the external system to call a PolicyCenter web service (SOAP) API 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` to the `ackMessage` method.

The following diagram illustrates the destination-related plugins, associated components, along with the chronological flow of actions between elements in the system.



Guidewire code
Your Code
External system

Note: A message transport can acknowledge the message from within the transport's `send()` method if possible. In that case, post-processing such as field updates would be done during the `send()` method. For destinations where the reply must be asynchronous (delayed), that destination must define a *message reply plugin* and its *callback block* to listen for and report the acknowledgement.

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.

Messaging Database Transactions During Sending

As mentioned in “Database Transactions When Creating Messages” on page 301, all steps up to and including adding the message to the send queue occurs in one database transaction. This is the same database transaction that triggered the event.

Additionally, there are special rules about database transactions during message sending at the destination level.

1. PolicyCenter calls `MessageRequest.beforeSend(...)` in one database transaction and commits changes assuming no exceptions occurred.
2. PolicyCenter calls `MessageTransport.send(...)` and then `MessageRequest.afterSend(...)` in one database transaction and commits changes assuming no exceptions occurred.
3. The `MessageReply` plugin, which optionally handles asynchronous acknowledgements to messages, does its work in a separate database transaction and commits changes assuming no exceptions occurred.

Message and Entity Locking

At the start of each transaction, PolicyCenter locks the message object itself at a database level. This ensures that the application does not try to acknowledge a skipped message or similar conditions. If some code tries to access the message while it is locked, the calling code waits for the current lock holder to unlock the message. See the diagram earlier in this section for the timing of messaging database transactions.

PolicyCenter can optionally lock the primary entity instance for a message if there is an associated safe ordered object for the message. The entity instance locking only affects the one entity instance, not any subobjects.

Similar to message locking, if some code tries to access the message while it is locked, the calling code waits for the current lock holder to unlock the message. PolicyCenter checks the `config.xml` parameter

`LockPrimaryEntityDuringMessageHandling`. If it is set to `true`, PolicyCenter locks the primary entity instance for that destination during the following operations:

- during send
- during message reply handling
- while marking a message as skipped

For example, if the message is associated with a policy, during all of these operations you can request that PolicyCenter optionally lock the associated `Policy` object at the database level. This can reduce problems in edge cases in which other threads try to modify objects associated with this policy.

WARNING This entity instance (and message) lock handling is separate from the concurrent data exception system. Be aware that concurrent data exception checking is disabled during messaging access. For important related information, see “Restrictions on Entity Data in Messaging Rules and Messaging Plugins” on page 299.

Built-In Destinations

Your rule sets can send standard e-mails and optionally attach the email to the policy as a document. The built-in email APIs use the built-in email destination. For more information about email configuration and API to send emails, see “Sending Emails” on page 95 in the *Rules Guide*. PolicyCenter always creates the built-in email destination, independent of your configuration settings.

Filtering Events

This section focuses on how PolicyCenter recognizes events and how to decide whether to notify external systems of these events.

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. Your validation rules set the validation level** – There is one rule set that governs validation checking, called Policy Validation Rules. These rules can set the validation level. Set the property `claim.ValidationLevel`.
- 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. Such a rule in Gosu might look like:

```
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.

List of Messaging Events in PolicyCenter

PolicyCenter generates events associated with a specific entity instance as the root object for the event. For more information about root objects, see “Root Object” on page 291. Also, contrast this with the concept of a primary object, see “Primary Entity and Primary Object” on page 291.

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 objects 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.

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, 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.
	ResyncAccount	Resync an account. This is an administrator request to drop all pending and <i>in error</i> 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. For more information, see "Resynchronizing Messages for a Primary Object" on page 339.
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.
	JobAdded JobChanged JobRemoved	Standard events for root entity Job.
Audit	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.
	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.
	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
	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.
Cancellation	CancellationAdded CancellationChanged CancellationRemoved	Standard events for root entity Cancellation.
	CancelPeriod	A period needs to be cancelled in the billing system. This is a custom event that is part of the PolicyCenter integration with BillingCenter.
Document	DocumentAdded DocumentChanged DocumentRemoved	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.
Job	JobAdded JobChanged JobRemoved	Jobs entities only exist as subtypes of Job, such as Audit. Those subtypes generate standard events for root entity Job.
	JobPurged	The job was removed through quote purging.
	RequestQuote	This is an important PolicyCenter event. PolicyCenter triggers this event to send a message to an external rating system to request a quote for a particular policy revision. See the “Rating Integration” on page 353 for details.
Letter	LetterAdded LetterChanged LetterRemoved	Standard events for root entity Letter.
Note	NoteAdded NoteChanged NoteRemoved	Standard events for root entity Note.
Organization	OrganizationAdded OrganizationChanged OrganizationRemoved	Standard events for root entity Organization.
Policy	PolicyAdded PolicyChanged PolicyRemoved	Standard events for root entity Policy. Most events are not on the policy entity but on an individual policy period entity.
	PolicyPurged	The policy was removed through quote purging.
	TransferPolicy	A policy transferred. This is a custom event used only by the PolicyCenter integration with BillingCenter.
PolicyPeriod	PolicyPeriodAdded PolicyPeriodChanged PolicyPeriodRemoved	Standard events for root entity PolicyPeriod.
	PolicyPurged	The policy period was removed through quote purging.
PolicyPeriodWorkflow	PolicyPeriodWorkflowAdded PolicyPeriodWorkflowChanged PolicyPeriodWorkflowRemoved	Standard events for root entity PolicyPeriod.
	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 finish and fail. For example, call <code>Submission.finishIssue(PolicyPeriod)</code> . Or, you can call <code>workflowinvokeTrigger(WorkflowTriggerKey)</code> . Workflow actions during acknowledgement are optional. The workflow does not automatically move forward due to an acknowledgement.

Entity	Events	Description
	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 acknowledgement are optional. The workflow does not automatically move forward due to an acknowledgement.
	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 acknowledgement 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 acknowledgement are optional. The workflow does not automatically move forward due to an acknowledgement.
	IssueReinstatement	A message fired to notify an external system to issue a reinstatement for a particular policy revision. The messaging plugin that submits the acknowledgement 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, <code>Submission.finishIssue(PolicyPeriod)</code> . Or, you can call <code>workflowInvokeTrigger(WorkflowTriggerKey triggerKey)</code> . Workflow actions during acknowledgement are optional. The workflow does not automatically move forward due to an acknowledgement.
	IssueRenewal	A message fired to notify an external system to issue a renewal for a particular policy revision. The messaging plugin that submits the acknowledgement 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 acknowledgement are optional. The workflow does not automatically move forward due to an acknowledgement.
	IssueSubmission	A message fired to notify an external system to issue a submission for a particular policy revision. The messaging plugin that submits the acknowledgement 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 acknowledgement are optional. The workflow does not automatically move forward due to an acknowledgement.
	SendCancellationNotices	Tell an external system to send cancellation notices.

Entity	Events	Description
	SendCancellationToSOR	Tell an external “System of Record” (SOR) system to send cancellation notices. The messaging plugin that submits the acknowledgement 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 acknowledgement are optional. The workflow does not automatically move forward due to an acknowledgement.
	SendCondRenewalDocuments	Tell an external system to send conditional-renewal documents. The messaging plugin that submits the acknowledgement 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 acknowledgement are optional. The workflow does not automatically move forward due to an acknowledgement.
	SendNonRenewalDocuments	Tell an external system to send non-renewal documents. The messaging plugin that submits the acknowledgement 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 acknowledgement are optional. The workflow does not automatically move forward due to an acknowledgement.
	SendNonRenewal	Tell an external system to send non-renewal data.
	SendNotTakenDocuments	Tell an external system to send not-taken-related documents. The messaging plugin that submits the acknowledgement 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 acknowledgement are optional. The workflow does not automatically move forward due to an acknowledgement.
	SendNotTakenToSOR	Tell an external “System of Record” (SOR) system about a not-taken-related document. The messaging plugin that submits the acknowledgement 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> . Although workflow actions during acknowledgement are optional, the workflow does not automatically progress forward from the acknowledgement.
	SendReinstatementNotices	Tell an external system to send reinstatement notices. The messaging plugin that submits the acknowledgement 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 acknowledgement are optional. The workflow does not automatically move forward due to an acknowledgement.

Entity	Events	Description
	SendRenewalDocuments	Tell an external system to send renewal-related documents. The messaging plugin that submits the acknowledgement 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 acknowledgement are optional. The workflow does not automatically move forward due to an acknowledgement.
	SendRescindNotices	Tell an external system to send rescind cancellation notices. The messaging plugin that submits the acknowledgement 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 acknowledgement are optional. The workflow does not automatically move forward due to an acknowledgement.
Issuance	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.
PolicyChange	PolicyChangeAdded PolicyChangeChanged PolicyChangeRemoved	Standard events for root entity PolicyChange.
	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.
ProducerCode	ProducerCodeAdded ProducerCodeChanged ProducerCodeRemoved	Standard events for root entity ProducerCode.
Reinstatement	ReinstateAdded ReinstateChanged ReinstateRemoved	Standard events for root entity Reinstate.
	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.
Renewal	RenewalAdded RenewalChanged RenewalRemoved	Standard events for root entity Renewal.
	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.
Rewrite	RewriteAdded RewriteChanged RewriteRemoved	Standard events for root entity Rewrite.
	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.
SubmissionBatch	SubmissionBatchAdded SubmissionBatchChanged SubmissionBatchRemoved	These are the standard events for root entity SubmissionBatch. SubmissionBatchAdded is an important event for PolicyCenter integration. PolicyCenter triggers this event if a submission batch submits and it is ready to go to a remote server for processing. In other words, every time one or more Submission objects submits using the Submission Manager, this event triggers.

Entity	Events	Description
Submission	JobAdded JobChanged JobRemoved	Standard events for root entity Job.
	BindSubmission	Fired to instruct an external system to bind a submission for a particular policy revision.
	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.
UnderwritingGroup	UnderwritingGroupAdded UnderwritingGroupChanged UnderwritingGroupRemoved	Standard events for root entity UnderwritingGroup.
Workflow	WorkflowAdded WorkflowChanged WorkflowRemoved	Standard events for root entity Workflow.
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 <code>processHistory.CompletionTime</code> property for the timestamp in a <code>datetime</code> object. For more information about batch processes, see “Batch Processing” on page 111 in the <i>System Administration Guide</i> .
SOAPCallHistory	SOAPCallHistoryAdded SOAPCallHistoryChanged SOAPCallHistoryRemoved	Standard events for SOAPCallHistory 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.
InboundHistory	InboundHistoryAdded InboundHistoryChanged InboundHistoryRemoved	Standard events for root entity Invoice.
Administration events		
Group	GroupAdded GroupChanged GroupRemoved	Standard events for root entity Group.
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.
GroupUser	GroupUserAdded GroupUserChanged GroupUserRemoved	Standard events for root entity GroupUser.
UserRoleAssignment	UserRoleAssignmentAdded UserRoleAssignmentChanged UserRoleAssignmentRemoved	Standard events for root entity UserRoleAssignment.

Entity	Events	Description
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.
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 ContactRemoved	Standard events for root entity Contact.

At What Time Does a Remove-related Event Trigger?

The *EntityNameRemoved* events trigger after either of the following occurs:

- some code deletes an entity from PolicyCenter
- some 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 retrievable. Refer to the *Data Dictionary* for details.

Triggering Custom Event Names

Business rules can trigger custom events using the `addEvent` method of policy entities and most other entities. This method can also be called from Java code that uses the Java API libraries, specifically from Java plugins or from Java classes called from Gosu.

You might choose to create custom events in response to actions within the PolicyCenter application user interface or using data changes originally initiated from the web service APIs. Triggering custom events is useful also if you want to use event-based rules to encapsulate code that logically represents one important action that could generate events. You could handle the event in your Event Fired rules but trigger it from another rule set such as validation, or from PCF pages.

To raise custom events within Gosu, use the `addEvent` method of the relevant object. In other words, call `myEntity.addEvent(eventname)`. In this case, the event name is whatever `String` you pass as the parameter. The entity whose `addEvent` method you call is the root object for the event.

You might also want to trigger custom events during message acknowledgement. If acknowledging the message from a messaging plugin, use the entity `addEvent` method described earlier.

Custom Events From SOAP Acknowledgements

Integrations that use SOAP API to acknowledge a message can use a separate mechanism for triggering custom events as part of the message acknowledgement.

First, your web service API client code in Java creates a new SOAP entity `Acknowledgement`. Next, call its `setCustomEvents` method to store a list of custom events to trigger as part of the acknowledgement:

```
String[] myCustomEvents = {"ExternalABC", "ExternalDEF", "ExternalGHI"};
myAcknowledgement.setCustomEvents(myCustomEvents);
```

Then, submit the acknowledgement using the SOAP APIs:

```
messagingToolsAPI.acknowledgeMessage(myAcknowledgement);
```

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 Tool

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. Remember that destinations signal which events they care about in the Messaging editor in 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. Because message creation impacts user response times, avoid unnecessarily large or complex messages if possible.

The most important object your Event Fired rules use is a *message context object*, which you can access 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. The following sections explain how to use business rules to analyze the event and generate messages.

WARNING Event Fired rules and messaging plugin implementations have limitations about what data you can change. See “[Restrictions on Entity Data in Messaging Rules and Messaging Plugins](#)” on page 299.

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. For more details, see “[Creating XML Payloads Using Guidewire XML \(GX\) Models](#)” on page 322.

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 Event Fired rule sets. For example, to check the destination ID number, use code like the following:

```
// 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 convert (*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)
```

Message Payloads and Setting Message Root or Primary Entities

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.

For more information about these concepts, see:

- “Root Object” on page 291
- “Primary Entity and Primary Object” on page 291
- “Setting a Message Root Object or Primary Object” on page 321

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. For example:

```
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:

```
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 policy.

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`.

You might need to share information across multiple runs of the Event Fired rule set for the same event or different events. If so, see “Saving Intermediate Values Across Rule Set Executions” on page 320.

Determining What Changed

In addition to normal access to the `messageContext`’s root object, there is a way to find out what has changed. Your Gosu business rule logic can determine which user made the change, the timestamp, and the original value of changed properties. This information is available only at the time you originally generate the message (*early binding*).

Note: You cannot use the `messageContext` object during processing of late bound properties, which are properties that employ *late binding* immediately before sending. For more information about late binding, see “Late Binding Data in Your Payload” on page 328.

At the beginning of your code, use `isFieldChanged` method test whether the property changed. If the field changed (and only if the property changed), call the `getOriginalValue` function to get the original value of that property. To get the new (changed) value, simply access the property directly on an entity. The new value has not yet been committed to the database. There are additional functions similar to `isFieldChanged` and `getOriginalValue` that are useful for array properties and other situations. Refer to “Determining What Data Changed in a Bundle” on page 344 in the *Gosu Reference Guide* for the complete list.

For example, the following Event Fired rule code checks if a desired property changed, and checks its original value also:

```
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
}
```

For a complete list of Gosu methods related to finding what data changed, see “Determining What Data Changed in a Bundle” on page 344 in the *Gosu Reference Guide*.

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.

Saving Intermediate 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 in all messages. You cannot save that 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.

There are two versions of this API, which are methods on the object returned by the Gosu expression `messageContext.SessionMarker`. Both APIs create a hash map that exists for multiple Event Fired rules executing in a single database transaction triggered by the same system event. However, there is an important difference:

- 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:

```
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:

```
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. For example:

```
var findResult = mycompany.QueryUtils.findRelatedObject().AtMostOneRow /* your own database query */
var resultToAdd = (findResult == null) ? null : messageContext.Bundle.add(findResult)
messageContext.SessionMarker.addToTempMap("MyKey", resultToAdd)
```

Creating a Payload 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 what 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 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 from within business rules to create some or all of your message payload.

For example, suppose you create a template file `NotifyAdminTemplate.gst` within the package `mycompany.templates`. Your fully-qualified name of the template is `mycompany.templates.NotifyAdminTemplate`.

Use the following code to generate (run) 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)
```

This code assumes the template supports parameter passing. For example, something like this:

```
<%@ params(myPolicyParameter : Policy) %>  
The Policy Number is <%= myPolicyParameter.PolicyNumber %>
```

There are a couple of steps. First, select the template. Then, you let templates use objects from the template's Gosu context using the `template.addSymbol` method. Finally, you 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.

For more information about using templates, see “Gosu Templates” on page 353 in the *Gosu Reference Guide*.

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. For further definition of a root object, see “Root Object” on page 291. Contrast this with the definition of primary object on “Primary Entity and Primary Object” on page 291.

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. For example:

```
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. See “Safe Ordering” on page 292 and “Message Ordering and Performance Tuning Details” on page 325.

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. As mentioned in “Primary Entity and Primary Object” on page 291, 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:

- `message.Account` – The account, if any, associated with this `Message` object.
- `message.Contact` – The contact, if any, associated with this `Message` object.

Additionally, there is a `message.PolicyPeriod` property. However, `PolicyPeriod` is not a primary entity for PolicyCenter messaging.

In PolicyCenter, if you set the `message.MessageRoot` property, the following behavior occurs automatically as a side-effect of setting this property from Gosu or Java:

- If the entity type is `Account` or has an `Account` property, PolicyCenter automatically sets the `message.Account` property to set the primary entity.
- If the entity type is `Contact` or has a `Contact` property, ClaimCenter automatically sets the `message.Contact` property to set the primary entity.
- Additionally, if the entity type is a `PolicyPeriod` or has a `PolicyPeriod` property, PolicyCenter automatically sets the `message.PolicyPeriod` property, however this 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 custom behavior of the message ordering system (safe ordering) by manually overriding properties that reference a primary object:

- Set the alternative primary entity in the messaging destination if you are not using an application default primary entity.
- In your Event Fired rules, manually set the primary entity property on the `Message` object that matches the primary entity for that destination. The primary entity for the destination is either the application default primary entity or overridden in the messaging destination configuration. You can set a primary entity property to `null` instead of an object. If the property for the primary entity for that destination is `null`, then PolicyCenter treats the message as a *non-safe-ordered* message. PolicyCenter orders safe-ordered and non-safe-ordered messages differently. For details, see “Message Ordering and Multi-Threaded Sending” on page 323.

WARNING 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.

Creating XML Payloads Using Guidewire XML (GX) Models

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 use this model to export XML or import XML in 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 its payload from an external system or return XML as its result.

The output XML only includes the properties specified in your custom XSD. It is best to create a custom XSD for each integration. Part of this is to ensure you send only the data you need for each integration point. For example, a check printing system probably needs a smaller subset of object properties than a external legacy financials system might need.

The first step is to create a new XML model in Studio. In Studio, navigate in the resource tree to the package hierarchy in which you want to store your XML model. Next, right-click on the package and from the contextual menu choose **New → Guidewire XML Model**.

For instructions on using the GX modeler, see “The Guidewire XML (GX) Modeler” on page 306 in the *Gosu Reference Guide*.

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. See “Calling Java from Gosu” on page 123 in the *Gosu Reference Guide*.

Saving Attributes of the 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 know how to handle the message. Alternatively, your destination could report on how many messages of each type were processed by PolicyCenter (for example, 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 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 acknowledgements. For example, to save a `Note` entity by the name `note1` to update a property on it later, use code similar to the following:

```
msg.putEntityByName("note1", note)
```

For more information about using `putEntityByName`, see “Reporting Acknowledgements and Errors” on page 329.

These methods are especially helpful to handle special actions in acknowledgements. 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 you messaging plugins acknowledge the message. The `getEntityByName` method always returns the correct reference to the original entity.

Maximum Message Size

In this version of PolicyCenter, messages can contain up to one billion characters.

Message Ordering and Multi-Threaded Sending

This section explains in detail how PolicyCenter orders messages and in some cases uses multiple threads.

For an overview of safe ordering, see the following topics:

- “Primary Entity and Primary Object” on page 291
- “Safe Ordering” on page 292
- “Setting a Message Root Object or Primary Object” on page 321

The application waits for an acknowledgement before processing the next safe-ordered message for that primary object for each destination. This allows PolicyCenter to send messages as soon as possible and yet prevent errors that might occur if related messages send out of order.

For example, suppose some external system must process an initial new message for a parent object before receiving any messages for subobjects or notes that relate to the parent object. If the external system rejects the new message for the parent object, it is not safe to send further messages to that system. However, it is likely safe to send messages about unrelated objects, or messages about the same object but to a different destination.

Even if the transport layer guarantees delivery order, it is unsafe for PolicyCenter to send the second message before confirming that the first safe-ordered message succeeds. Doing otherwise could cause difficult error recovery problems.

Safe ordering has large implications for messaging performance. For a destination configured with Account as the primary entity, suppose the send queue contains 10 messages associated with different accounts, PolicyCenter can send these 10 safe-ordered messages immediately. However, if the send queue contains 10 safe-ordered messages for the same account, PolicyCenter can only send one. PolicyCenter must wait for the message acknowledgement, and then at that point can send the next (only one) message for that account. Another way of thinking about is that only one message can be in flight for each account/destination pair.

If you license the ContactManager application for use with PolicyCenter, note that ContactManager also supports safe ordering of messages associated with each ABContact entity. This means that ContactManager only allows one message for each combination of ABContact and destination pair at any given time. For more ContactManager integration information, see “ContactManager Integration Reference” on page 231 in the *Contact Management Guide*.

Ordering Non-safe-ordered Messages

Some messages are not associated with a primary object such as Account (or Contact, if a destination specifies Contact as the alternative primary object). These cross-account messages are called *non-safe-ordered messages*.

These messages are sometimes also called Messages Without Primary.

See the beginning of this topic for links to topics that define primary entities and safe ordering.

By default, non-safe-ordered messages send as soon as possible, before any queued safe-ordered messages, and send in the order that Event Fired rules generated them. By default PolicyCenter does not wait for acknowledgements for non-safe-ordered messages before sending the next -safe-ordered message.

For example, PolicyCenter sends a message about a new User immediately. PolicyCenter never waits for acknowledgements for other messages before sending this message.

However, if you enable the Strict Mode feature for a destination in the Messaging editor in Studio, PolicyCenter waits for the acknowledgement for every non-safe-ordered message. There are also destination options for multi-threaded sending of non-safe-ordered messages. For details, see “How Destination Settings Affects Ordering” on page 327.

For PolicyCenter, messages for the following events are by default non-safe-ordered messages:

- Group events
- User events

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. For more information about destination setup, see “Message Destination Overview” on page 302.

If you register a messaging plugin, you must register it in two places. First, register it in the plugin registry in the plugin editor. See “Using the Plugins Registry Editor” on page 131 in the *Configuration Guide*. Next, register it in the messaging registry in the messaging editor; see “Using the Messaging Editor” on page 153 in the *Configuration Guide*.

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. Refer to “Message Ordering and Multi-Threaded Sending” on page 323 for details.

Message Ordering and Performance Tuning Details

PolicyCenter pulls messages from the database (the send queue) in batches of messages on the batch server only, and then waits for a polling interval before querying again.

You can configure the number of messages that the messaging subsystem retrieves in each round of sending. This is called the chunk size. Configure the chunk size in the messaging destination configuration editor in the **Chunk Size** field. By default, this value is set to 100,000, which typically includes all sendable messages currently in the send queue. You can also change the polling interval in messaging destination configuration editor in the **Polling Interval** field.

You must understand the difference between message readers and message sending threads:

- *Message readers* are threads that query the database for messages. Message readers use the *message send order* (typically this is equivalent to creation order). The message reader never loads more than the maximum number of messages in the chunk size setting at one time.
- *Message sender threads* are threads that actually call the messaging plugins to send the messages. A new feature in this release is support for multiple sender threads per messaging destination for safe-ordered messages. You can configure the number of sender threads for safe-ordered messages in the messaging destination configuration editor in the **Number Sender Threads** field.

The messaging ordering and sending architecture works as follows by default (see “How Destination Settings Affects Ordering” on page 327):

1. Each messaging destination has a worker thread that queries the database for messages for that destination only. In other words, *each destination has its own message reader*. Each message reader thread (each worker thread) acts independently.
2. The destination’s message reader queries the database for one batch of messages, where the maximum size is defined by the chunk size. PolicyCenter does not use the chunk size value in the query itself. Instead, the message reader orders the results by send order and stops iterating across the query results after it retrieves that many messages from the database.

PolicyCenter performs two separate queries:

- a. First, PolicyCenter queries for messages associated with a primary object for that destination, also known as *safe-ordered messages*. The maximum number of messages returned for this query is also the chunk size. The database query itself ensures PolicyCenter that no more than one message per primary object exists in this list. If another message for a primary object is sent but unacknowledged, no messages for that object appears in this list. This enforces the rule that no more than one message can be in-flight for each primary object per destination.

The query ensures that no two messages are in flight (sent but not yet acknowledged) for the same destination for the same primary object. So, if there are 100 messages for one primary object, the query only reads and dispatches one of those messages (out of a possible 100) to the destination subthreads.

- b. Next, PolicyCenter queries for all messages not associated with the primary object for that destination, also known as *non-safe-ordered messages*. If the chunk size is not set high enough, the returned set is not the full set of non-claim-specific messages. Be aware the chunk size affects each query. It is not cumulative for safe-ordered and non-safe-ordered messages.

By default, the chunk size is set to 100,000, which is usually sufficient for customers. Be sure not to lower the chunk size too much. Typically there are dependencies between safe-ordered and non-safe-ordered messages on that destination. If the chunk size is too low, the non-safe-ordered message query might not retrieve all the non-safe-ordered messages that your safe-ordered messages rely upon. The downstream system might need to receive the message that applies to many primary objects before any other messages reference that information.

3. For each destination, the worker thread iterates through all non-safe-ordered messages for that destination, sending to the messaging plugins. Settings in the Messaging editor in Studio for each destination affect how non-safe-ordered messages are sent. The choices are single thread, multi-thread, or strict mode. Those settings affect how many threads send messages, and how the application handles errors. See “How Destination Settings Affects Ordering” on page 327.

After each worker thread finishes sending non-safe-ordered messages, it creates subthreads to send safe-ordered messages for that destination. Configure the number of threads in the messaging destination configuration editor in the **Number Sender Threads** field. Each worker thread distributes the list of safe-ordered messages to send to the subthreads.

Assigning the number of sender subthreads for a destination by default affects only the safe-ordered messages for that destination. For non-safe-ordered messages (also called Messages Without Primary), the rules depend on destination settings. See “How Destination Settings Affects Ordering” on page 327.

If the message is associated with a primary entity, during messaging operations you can optionally lock the primary entity at the database level. This can reduce some problems in edge cases in which other threads (including worker threads) try to modify objects associated with this same object.

For example, if using the default Account primary entity, the system locks the associated Account during messaging.

PolicyCenter checks the `config.xml` parameter `LockPrimaryEntityDuringMessageHandling`. If it is set to `true`, PolicyCenter locks the primary entity during message send, during all parts of message reply handling, and while marking a message as skipped.

4. The message reader thread waits until all destination threads send all messages in the queues for each subthread.
5. PolicyCenter checks how much time passed since the beginning of this round of sending (since the beginning of step 2) and sleeps the remainder of the polling interval. Configure the polling interval in the messaging destination configuration editor in the **Polling Interval** field. If the amount of time since the last beginning of the polling interval is `TIME_PASSED` milliseconds, and the polling interval is `POLLING_INTERVAL` milliseconds. If the polling interval since the last query has not elapsed, the reader sleeps for `(POLLING_INTERVAL - TIME_PASSED)` milliseconds. If the time passed is greater than the polling interval, the thread does not sleep before re-querying.

The message reader reads the next batch of messages. Begin this procedure again at step 2.

The polling interval setting critically affects messaging performance. If the value is low, the message reader thread sleeps little time or even suppresses sleeping between rounds of querying the database for more messages.

To illustrate how this works, compare the following situations.

Suppose there are two messaging destinations, and both destinations use Account as the primary entity. Also suppose the send queue contains 10 messages for each destination. For each destination, assume that there is no more than one message for each account. In other words, for each destination, there are 10 total messages related to 10 different accounts:

- Assuming the number of messages does not exceed the chunk size, each destination gets only one message for the account for that destination from the database. In this case, every message can be sent immediately because each message for that destination is independent because they are for different accounts. If the destination's **Number Sender Threads** setting is greater than 1, PolicyCenter distributes all account-specific messages to multiple subthreads. The length of the destination queue never exceeds the number of messages queried in each round of sending. The message reader waits until all sending is complete before repeating.

In contrast, suppose the messages for one destination includes 10 account-specific messages for the same account and 5 non-account-specific messages:

- Assuming the number of messages does not exceed the chunk size, PolicyCenter reads all 5 non-safe-ordered messages and sends them. However, PolicyCenter only gets one message for the account for that destination from the database. If the destination's **Number Sender Threads** setting is greater than 1, PolicyCenter distributes all account-specific messages in multiple threads per destination. In this case there is only message that is sendable. Compared to the previous example, PolicyCenter handles fewer messages for each polling interval.

To improve performance, particularly cases like the second example, change the following settings in the Messaging editor for your destinations:

- Lower the **Polling Interval**. The value is in milliseconds. Experiment with lower values perhaps as low as 1000 (which means 1 second) or even lower. Test any changes to see the real-world effects on your messaging performance. If your performance issues primarily relate to many messages per primary entity per destination, then the polling interval is the most important messaging performance setting.
- Increase the value for **Number Sender Threads**. This permits more worker threads to operate in parallel on the batch server only for sending safe-ordered messages. Again, test any changes to see the real-world effects on your messaging performance. If your performance issues primarily relate to many messages but few messages per primary entity for each destination, then the sending threads number is the most important messaging performance setting.

To get maximum performance from multiple threads, keep your message transport plugin implementation's `send` method as quick as possible, with little variation in duration. The application does not load the next chunk of messages until the application passes all messages in the chunk to the message transport `send` method. If your `send` method sometimes takes a long time, convert your code to asynchronous sending. With asynchronous sending, the `send` method completes quickly and the reply is handled later. See "Message Destination Overview" on page 302.

Thread-Safe Plugins

You must write all messaging plugin implementation code as *thread-safe code*. This means that you must be extremely careful about static variables and any other shared memory structures that multiple threads might access running the same (or related) code.

For more information, see "Concurrency" on page 375 in the *Gosu Reference Guide*.

You must write your messaging plugin code as thread-safe even if the **Number Sender Threads** setting is set to 1.

How Destination Settings Affects Ordering

For messages without a primary object (sometimes called non-safe-ordered messages), there are settings in the Messaging editor for each messaging destination that configures how to order them.

The **Message Without Primary** setting has three choices:

- Single thread** – Messages without a primary object send in a single thread, and do not wait for an acknowledgement before proceeding to other messages.
- Multi thread** – Messages without a primary object send in multiple threads, and do not wait for an acknowledgement before proceeding to other messages. The precise order of sending of messages without a primary object is non-deterministic.

- **Strict Mode** – If Strict Mode is enabled, messages without a primary object send in a strict order, and wait for an acknowledgement before proceeding to other messages.

Carefully choose the value for each messaging destination.

WARNING If you use either **Single thread** or **Multi thread** options, errors for messages without a primary object do not hold up other messages. This means that errors in such messages may cause errors at the destination if the system is unprepared for this situation.

If Strict Mode is enabled, if errors occur other messages for that destination stop until an administrator resolves the problem. For example, an administrative user can resync that message. Resyncing the messages allows other messages to send for that primary object for that destination after resynchronizing that primary object. For more information about resynchronizing, see “Resynchronizing Messages for a Primary Object” on page 339.

With the value to **Single thread** or **Multi thread**, which means Strict Mode is disabled:

- Each destination sends non-safe-ordered messages in one or more threads, depending on whether you choose **Single thread** or **Multi thread**. Next, the application sends safe-ordered messages.

Note: Optionally, sending safe-ordered messages is multi-threaded. For more, see “Message Ordering and Multi-Threaded Sending” on page 323.

- Non-safe-ordered messages send in order but never wait for acknowledgement. Non-safe-ordered message errors never block other messages.

With Strict Mode on, the behavior is:

- Non-safe-ordered messages require an acknowledgement before sending future next message (of either type) to that destination. This option reduces throughput of non-safe-ordered messages.
- Delay sending safe-ordered messages until all non-safe-ordered messages are sent.
- Errors block all future messages for that destination, independent of the message type.

For each messaging destination, carefully consider the data integrity, error handling, and performance needs for your system before deciding on the value of the Message without Primary option.

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 includes any changes in this database transaction (entity instance additions, removals, or changes). Generally this is the best approach for messaging. If you waited until messaging sending time (rather than creation time), the data might be partially different or even data removed from the database. This could 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 is the standard recommended approach. This is called *early binding*.

However, this prevents *later* changes to an object appearing in an *earlier* message about that object. This is relevant if there is a large delay in sending the message for whatever reason. 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, imagine sending a new policy to an external system. As part of the acknowledgement, 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 policy's new public ID received from the external system in the acknowledgement. The external system knows which policy belongs with this second message. If the public ID were merely set during the original processing of the event, that message does 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.

IMPORTANT Guidewire recommends exporting all data in the Event Fired rules into a message payload (early-bound) unless you have a specific reason why late binding is critical for that situation.

For newly-created entity instances, some customers send the entity instance's *public ID property* as a late bound property. A message acknowledgement or external system (using web service APIs) could change the public ID between creating (submitting) the message and sending it.

For other properties, decide whether early binding or late binding is most appropriate.

At message creation time in Event Fired rules, add your own marker text within the message, for example "<AAAAAA>". Your `MessageRequest` plugin code or `MessageTransport` plugin code can directly find the message root object or primary object and substitute the current value. If PolicyCenter calls your `MessageRequest` plugin, the current value of the property is a late bound value and you can replace the marker with the new value.

For example, a Gosu implementation of the `MessageRequest` plugin interface might do this using 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  
}
```

See also

- “Primary Entity and Primary Object” on page 291
- “Setting a Message Root Object or Primary Object” on page 321

Reporting Acknowledgements and Errors

PolicyCenter expects to receive acknowledgements back from the destination. In most cases, the destination submits a *positive acknowledgement* to indicate success. However, errors can also occur, and you must tell PolicyCenter about the issue.

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 acknowledgements.

Error conditions during the destination `send` process:

- **Exceptions during send() causes automatic retries** – Sometimes a message transport plugin has a send error and expects it 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.
- **For errors during send() and you do not want automatic retry** – If the destination has an error that the application does not expect to be temporary, do not throw an exception. Throwing an exception triggers automatic retry. Instead, call the message's reportError method with no arguments. See “Submitting Acks, Errors, and Duplicates from Messaging Plugins” on page 330.

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.

Errors conditions that occur later:

- **A negative acknowledgement (NAK)** – A destination might get an error reported from the external system (database error, file system error, and 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 acknowledgement 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 whereas resending results in message duplicates. External system may 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 acknowledgement (ACK) or some sort of error (NAK).

For PolicyCenter administrative tools that monitor and recover from messaging errors, see “The Administration User Interface” on page 346.

Submitting Acks, Errors, and Duplicates from Messaging Plugins

To submit an acknowledgement or a negative acknowledgement from a messaging plugin implementation class, use the following APIs. Refer to the following table based on the success status, the type of failure, and the location of your code.

Situation	Do this	Description
Report acknowledgement		
Success	<code>message.reportAck()</code>	Submits an ACK for this message, which may permit other messages to be sent. For detailed logic of message ordering, see “Message Ordering and Multi-Threaded Sending” on page 323.
Errors within the send method of your <code>MessageTransport</code> plugin implementation, and the error is presumed temporary.	Throw an exception within the send method, which triggers automatic retries potentially multiple times.	<p>Automatically retries the message potentially multiple times, including the backoff timeout and maximum tries. After the maximum retries, the application ceases to automatically retry it and suspends the messaging destination.</p> <p>An administrator can retry the message from the Administration tab. Select the message and click Retry. For details of automatic retry, see “Message Sending Error Behaviors” on page 329.</p>

Situation	Do this	Description
Report error		
Errors in any messaging plugins and no automatic retry is needed	<code>message.reportError()</code>	<p>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.</p> <p>For more information about the retry schedule, see “Reporting Acknowledgements and Errors” on page 329.</p>
Errors in any messaging plugins and scheduled retry is needed	<code>message.reportError(date)</code>	<p>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.</p>
Errors in any messaging plugins and scheduled retry is needed	<code>message.reportError(category)</code>	<p>Reports the error and assigns an error category from the <code>ErrorCategory</code> typelist. The Administration tab uses this error category to identify the type of error in the user interface. You can extend this typelist to add your own meaningful values.</p> <p>In the base configuration of PolicyCenter, this typelist contains values such as:</p> <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 <p>You can call this method from the <code>MessageTransport</code> or when handling errors in your <code>MessageReply</code> plugin implementation.</p>

Situation	Do this	Description
Report duplicate		
Message is a duplicate	<code>msgHist.reportDuplicate()</code>	<p>Reports a duplicate. This is a method on the message history object, not the message object. A message history object is what a message becomes after successful sending. A message history (<code>MessageHistory</code>) 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> <ul style="list-style-type: none"> 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: <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. For related information about asynchronous replies, see "Implementing a Message Reply Plugin" on page 336</p>

Using Web Services to Submit Ackcs and Errors From External Systems

If you want to acknowledge the message directly from an external system, use the web service method `IMessagingTools.acknowledgeMessage(ack)`. First, create an `Acknowledgement` SOAP object.

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, always set the `Acknowledgement.Retryable` property to `true` and `Acknowledgement.Error` to `true`. Set this property to `false` (the default) only if there is no error.
- **Duplicate** – If you detect the message is a duplicate, set the `Duplicate` and `Error` properties to `true`.

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 acknowledgement, 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.

Tracking a Specific Entity With a Message

If desired, 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 acknowledgements. To attach an entity to a message in Event Fired rules, use the `Message` method `putEntityByName`. to attach an entity to this message and associate it with a custom ID called a *name*. Later, as you process an acknowledgement, 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 acknowledgements. 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 changed. This is particularly useful if the public IDs on some objects changed between the time you create the message and the time the messaging code acknowledges the messaged. In such cases, `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 acknowledgement, 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 indicate how the external system thinks of this object.

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 acknowledgement.

Implementing Messaging Plugins

There are three types of messaging plugins. See the following sections for details:

- “[Implementing a Message Request Plugin](#)” on page 334
- “[Implementing a Message Transport Plugin](#)” on page 334
- “[Implementing a Message Reply Plugin](#)” on page 336

WARNING Event Fired rules and messaging plugin implementations have limitations about what data you can change. See “[Restrictions on Entity Data in Messaging Rules and Messaging Plugins](#)” on page 299.

Getting Message Transport 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 this:

```
class MyTransport implements MessageTransport {
```

Change it to this:

```
class MyTransport implements MessageTransport, InitializablePlugin {
```

To conform to this new interface, add a `setParameters` method with the following signature:

```
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
    var myValueFromTheMap = map["servername"]
}
```

Implementing a Message Request Plugin

A destination can optionally define a message request (`MessageRequest`) plugin to prepare a `Message` object before a message is sent to the message transport. It may not be necessary to do this step. For example, if textual message payload contains strings or codes that must be translated for a specific remote system. Or perhaps a textual message payload contains simple name/value pairs that must be translated into a XML before sending to the message transport. Or, perhaps you need to set data model extension properties on `Message`.

To prepare a message before sending, implement the `MessageRequest` method `beforeSend`. PolicyCenter calls this method with the message entity (a `Message`).

The main task for this method is to generate a modified payload and return it from the method. Generally speaking, do not modify the payload directly in the `Message` entity. The result from this method passes to the messaging transport plugin to send in its `transformedPayload` parameter. This transformed payload parameter is separate from the `Message` entity that the message transport plugin gets as a parameter.

You can modify properties within the `Message` or within the message's root object such as a `Policy` object as needed. However, as mentioned before, to transform the payload just return a `String` value from the method rather than modifying the `Message`.

You can also use the `MessageRequest` plugin to perform post-sending processing on the message. To perform this type of action, implement the `MessageRequest` method `afterSend`. PolicyCenter calls the method with the message (a `Message` object) as a parameter. PolicyCenter calls this method *immediately* after the transport plugin's `send` method completes. If you implement asynchronous callbacks with a message reply plugin, be sure to understand that the server calls `afterSend` in the same thread executing the plugin's `send` method. However, it might be a separate thread from any asynchronous callback code.

Also, if the `send` method acknowledges the message with any result (successful ACK or an error), then be aware that the ACK does not affect whether the application calls `afterSend`. Assuming no exceptions trigger during calls to `beforeSend` or `send`, PolicyCenter calls the `afterSend` method.

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.

Implementing a Message Transport Plugin

A destination must define a message transport (`MessageTransport`) plugin to send a `Message` object over some physical or abstract transport. This might involve submitting a message to a message queue, calling a remote web service API, or might implement a complex proprietary protocol specific to some remote system. The message transport plugin is the only required plugin interface for a destination.

To send a message, implement the `send` method, which PolicyCenter calls with the message (a `Message` object). That method has another argument, which is the *transformed payload* if that destination implemented a message request plugin. See "Implementing a Message Request Plugin" on page 334.

In the message transport plugin's simplest form, this 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. Your `send` method optionally can immediately acknowledge 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 may optionally update properties on PolicyCenter objects such as `Policy`. If you desire this, 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. To visualize how these elements interact, see the diagram in “Message Destination Overview” on page 302.

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 to handle the asynchronous reply. See the “Implementing a Message Reply Plugin” on page 336 for details.

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 acknowledgement for that message.

Your implementations of messaging plugins must explicitly implement `InitializablePlugin` in the class definition. This interface tells PolicyCenter that you support the `setParameters` method to get parameters from the plugin registry. Even if you do not need parameters, your plugin implementation must implement `InitializablePlugin` or the application does not initialize your messaging plugin.

Your implementation of this plugin must explicitly implement `InitializablePlugin` in the class definition and then also add a `setParameters` method to get parameters. Implement this interface even if you do not need the parameters from the plugin registry.

The following example in Java demonstrates a basic message transport.

Example Basic Message Transport 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)
        message.reportAck()
    }
}
```

More Examples

Several example message transport plugins ship with PolicyCenter in the product in the following directory:

`PolicyCenter/java-api/examples/src/examples/pl/plugins/messaging`

Exception Handling

For details of exceptions and handling suspect/shutdown in messaging plugins, see “Error Handling in Messaging Plugins” on page 338.

Implementing a Message Reply Plugin

A destination can optionally define a message reply (`MessageReply`) plugin to asynchronously acknowledge a Message. For instance, this plugin might implement a trigger from an external system that notifies PolicyCenter that the message send succeeded or failed.

PolicyCenter requires a special step in this process to setup the PolicyCenter database transaction information appropriately so that any entity changes commit to the PolicyCenter database. To do this properly, Guidewire provides the types `MessageFinder`, `PluginCallbackHandler`, and inner interface `PluginCallbackHandler.Block`.

A message finder (`MessageFinder`) object is built-in object that returns a `Message` entity instance from its `messageID` or `senderRefID`, using its `findById` or `findBySendRefId` method. During the message reply plugin initialization phase, PolicyCenter calls the message reply plugin `initTools` method with an instance of `MessageFinder`. Save this instance of `MessageFinder` in a private variable within your plugin instance.

A plugin callback handler (`PluginCallbackHandler`) is an object provided to your message reply plugin. The callback handler safely executes the message reply callback code block. The callback handler sets up the callback’s server thread and the database transaction information so entity changes safely and consistently save to the database.

If you want to get parameters from the plugin registry, your messaging plugin implementation must explicitly implement `InitializablePlugin` in its class declaration. The `InitializablePlugin` interface declares that the class supports the `setParameters` method to get parameters from the plugin registry. The application calls the plugin method `setParameters` and passes the parameters as name-value pairs of `String` values in a `java.util.Map` object.

For important information about using message finders to submit errors and duplicates, see “Error Handling in Messaging Plugins” on page 338.

IMPORTANT PolicyCenter includes multi-threaded 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`. See “Multi-threaded Inbound Integration Overview” on page 267 and “Custom Inbound Integrations” on page 278.

Message Reply Plugin Initialization

During initialization, the message reply plugin must get and store a reference to the message finder (`MessageFinder`) and callback handler (`PluginCallbackHandler`) objects, since it needs to use them later.

Implement the simple `MessageReply` interface and include the `initTools` method, which gets these objects as parameters. Your plugin implementation must store these values in private properties, for example in private properties `_pluginCallbackHandler` and `_messageFinder`.

Message Reply Callbacks

To acknowledge the message asynchronously, your message reply plugin uses its reference to the `PluginCallbackHandler`. To run your code, you plugin must pass a specially-prepared block of Java code to the `PluginCallbackHandler` method called `execute`.

The `execute` method takes a *message reply callback block*, which is an instance of `PluginCallbackHandler.Block`. The `Block` is a simple private interface to encapsulate the block of code that you write.

The `PluginCallbackHandler` object has an `add` method that marks a PolicyCenter entity as modified so the application commits changes to the database with the message acknowledgement. Call the `add` method and pass an entity instance. This method adds the entity to the correct *bundle*. For more about bundles, see “Bundles and Database Transactions” on page 337 in the *Gosu Reference Guide*. This process ensures that changes to the entity instance commit to the database in the correct database transaction with related changes.

Your code in `PluginCallbackHandler.Block` must perform the following steps:

1. Find a `Message` object. Use methods on the plugin’s `MessageFinder` instance, described earlier in this section.
2. Call methods on the `Message` to signal acknowledgement or errors:

- To report success, call the `reportAck` method on the message.
- To report errors, call the `reportError` method on the message. There are multiple method signatures to accommodate automatic retries and error categories.
- To report duplicates, call the `reportDuplicate` method on the message history (`MessageHistory`) entity instance that corresponds to the original message.

For details, see “Submitting Acks, Errors, and Duplicates from Messaging Plugins” on page 330.

3. Optional post-processing such as property updates or triggering custom events. If any objects must be modified other than the objects originally attached to the message, the callback block must call `PluginCallbackHandler.add(entityReference)`. This call to the `add` method ensures all changes on the other objects properly commit to the database as part of that database transaction. You must use the return result of the `add` method and only modify that return result. The return result is a clone of the object that is now writable. Do not continue to hold a reference to the original object you passed to the `add` method. This is equivalent to the `bundle.add(entityReference)` method, which adds an entity instance to a writable bundle. For more information, see “Adding Entity Instances to Bundles” on page 341 in the *Gosu Reference Guide*.

For example, the following example demonstrates creating a `PluginCallbackHandler.Block` object and executing the callback block.

```
...
PluginCallbackHandler.Block block = new PluginCallbackHandler.Block() {
    public void run() throws Throwable {
        Message message = _messageFinder.findById(messageID);
        message.reportAck();
    }
};

\PluginCallbackHandler.execute(block);

...
```

You can call `EntityFactory` as necessary in your callback handler block to create or find entities. The application properly sets up the thread context so that it supports `EntityFactory`. For more information about `EntityFactory`, see “Accessing Entity and Typecode Data in Java” on page 633.

For details of exceptions and handling suspend/shutdown in messaging plugins, see “Error Handling in Messaging Plugins” on page 338.

Error Handling in Messaging Plugins

For a summary of message error APIs, see “Reporting Acknowledgements and Errors” on page 329.

Several methods on messaging plugins execute in a strict order for a message. Consult the following list to design your messaging code, particularly error-handling code:

1. PolicyCenter selects a message from the send queue on the batch server.
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 `send` and `afterSend` methods (step 4 and step 5) commit if and only if no exceptions occurred yet. Any exceptions roll 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

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. See “Submitting Acks, Errors, and Duplicates from Messaging Plugins” on page 330.

Handling Duplicates

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. For more information about acknowledgements and errors, see “Submitting Acknowledgements and Errors” on page 330.

Saving the Destination ID for Logging or Errors

Each messaging plugin implementation must have a `setDestinationID` method, which allows the plugin to find out its own destination ID and store it in a private variable. The destination can use the destination ID within code such as:

- Logging code to record the destination ID.
- Exception-handling code that works differently for each destination.
- Other integrations, such as sending destination ID to external systems so that they can suspend/resume the destination if necessary.

Handling Messaging Destination Suspend, Resume, Shutdown

The standard messaging plugins have methods that perform special actions for the administrative commands for destination suspend, destination resume, and before messaging system shutdown.

Typically, suspend and resume is the result of an administrator using the PolicyCenter **Administration** tab in the user interface to suspend or resume a messaging destination. Alternatively, suspend and resume could be the result of a web service API call to suspend or resume destinations.

Messaging shut down occurs during server shutdown or if the configuration is about to be reread. After message sending resumes again, PolicyCenter reuses the same *instance* of the plugin. PolicyCenter does not destroy the existing messaging plugin instance (or recreate it) as part of shutdown.

To trap these actions, implement the `suspend`, `shutdown`, and `resume` methods of your messaging plugin.

You can implement these methods just for logging and notification, if nothing else. For example, a message transport plugin’s `suspend` or `shutdown` methods could log the action and send notifications as appropriate. For example:

```
public void suspend() {  
    ...  
  
    if(_logger.isDebugEnabled()) {  
        _logger.debug("Suspending message transport plugin.")  
    }  
  
    MyEmailHelperClass.sendEmail("....")  
}
```

During the `suspend`, `shutdown`, and `resume` methods of the plugin, the plugin must not call any PolicyCenter `IMessageToolsAPI` web service APIs that suspend or resume messaging destinations. Doing so creates circular application logic, so such actions are forbidden.

It is unsupported for a messaging plugin `suspend`, `shutdown`, and `resume` method to use messaging web service APIs that suspend or resume messaging destinations.

Resynchronizing Messages for a Primary Object

PolicyCenter implementations can use the messaging system to synchronize data with an external system.

In rare cases some messaging integration condition might fail, such as failure to enforce an external validation requirement properly. If this happens, an external system might process one or more PolicyCenter messages incorrectly or incompletely. If the destination detects the problem, the external system returns an error. The error must be fixed or there may be synchronization errors with the external system.

However, suppose the administrator fixes the data in PolicyCenter and improves any related code. It still might be that the external system has incorrect or incomplete data. PolicyCenter provides a programming hook called a *resync event* (a resynchronization event) to recover from such messaging failures.

To trigger this manually, an administrator navigates to the Administration tab in PolicyCenter, views any unsent messages, clicks on a row, and clicks **Resync**.

The resync can be triggered from web services using the `MessagingToolsAPI` web service method `resyncAccount`.

Although `Contact` can be an 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 (`resyncAccount`). Configure your messaging destination to listen for this event. Then, implement Event Fired business rules that handle that event.

Afterwards, PolicyCenter marks all messages that were pending as of the resync as *skipped*. You must implement Guidewire Studio rules that examine the 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 Gosu resync Event Fired rules to how your particular external systems recover from such errors.

There are two different basic approaches for generating the resync messages.

In the first approach, your Gosu rules traverse all claim 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 claim 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 may be necessary to add custom extension properties to various objects with the synchronization state. If you can somehow determine that you only need to resend a subset of messages, only send that minimal amount of information. However, one of the benefits of resync is the opportunity to send all information might conceivably be out of sync. Think carefully about 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 message and all queued and unsent messages for the claim for a specific destination. Your rules use that information to determine which messages to recreate. Instead of examining the entire claim's history you could consider only the failed and unsent messages. Because a message with an error prevents sending subsequent messages for that claim, there may be many unsent pending messages. To help with this process, PolicyCenter includes properties and methods in the rules context on `messageContext` and `Message`.

From within your Event Fired rules, your Gosu code can access the `messageContext` object. It 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 all these original pending messages. This means that 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 has length zero.

WARNING If you create new messages, the new messages send in creation order independent of the order of original messages. This might be a different order than the original messages. Think carefully about how this may or not affect edge cases in the external system.

There are various properties within any message 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 creates 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.p1.messaging.MessageStatus` contains static properties and static methods that you can use for easy to understand code. For a complete reference, see “Message Status Code Reference” on page 343.
- **ErrorDescription** - A string that contains the description of errors, if any. This may or may not be present. This is set within a negative acknowledgement (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 only need to use the sender reference ID 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 ID property populated by the destination.

Cloning New Messages From Pending Messages

As mentioned earlier, during resync you can clone a new message from a pending message that you got from `messageContext.PendingMessages`. To clone 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.

IMPORTANT All pending messages skip after the Event Fired rules for the resync event complete. You must create equivalent new messages for all pending messages.

How Resync Affects Pre-Update and Validation

Be aware that pre-update and validation rule sets do not run solely because of a triggered event. A policy's pre-update and validation rules run only if actual entity data changes. In cases where triggered events do not correspond to already-changed entities, the event firing alone does not trigger policy pre-update and validation rules.

This does not affect most events because almost all events correspond to entity data changes. However, for the events related to resync, no entity data inherently changes due to this event.

This also can affect other custom event firing through the `addEvent` entity method.

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. For more information, see “ContactManager Messaging Events by Entity” on page 242 in the *Contact Management Guide*.

Message Payload Mapping Utility for Java Plugins

A messaging plugin may need to convert items in the payload of a message before sending it on to the final destination. A common reason for this is mapping typecodes. For many properties governed by typelists, the typecode might have the same meaning in both systems. However, this does not work for all situations. Instead, you might 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 using text substitution. This class scans a message payload to find any strings surrounded by delimiters that you define and then substitutes a new value. The class is `com.guidewire.util.StringSubstitution`. Refer to the *Java API Documentation* for reference.

To use the String substitution Java class from Java plugin code

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.
2. Put these delimiters around the Gosu template text to map and replace. For example, a Gosu template might include “Injury=**exposure.InjuryCode**”. This might generate text such as “Injury=**A1**” in the message payload.
3. Implement a class that implements the inner interface called `StringSubstitution.Exchanger`. This exchanger class might use its own look-up table or look in a properties file and substitute new values. 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.
4. Instantiate your class that implements Exchanger, and then instantiate the `StringSubstitution` class with the constructor containing your delimiters and your Exchanger instance.

```
MyExchanger myExchangerInstance = new MyExchanger()  
StringSubstitution mySub = new StringSubstitution(" ** ", " ** ", myExchangerInstance)
```

5. Call the `substitute` method on the `StringSubstitution` instance to convert the message payload string.

Message Status Code Reference

The following table describes the message status values and the contexts they can appear for `Message` and `MessageHistory` objects. The second column indicates the static property you can use on the `gw.pl.messaging.MessageStatus` class to make your code easier to understand.

The status code information is particularly valuable for Gosu code that implements resynchronization (*resync*). See “Resynchronizing Messages for a Primary Object” on page 339.

MessageStatus static property	Message status value	Meaning	Valid in Message	Valid in MessageHistory	Can appear during resync
PENDING_SEND	1	Pending send. this is the initial state for messages.	●		●
PENDING_ACK	2	Messages 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 This state is sometimes referred to as <i>in flight</i> .	●		●
ERROR	3	<i>Legacy non-retryable error. Provided for legacy use and is no longer used.</i>			
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. This is the original message as represented as a <code>MessageHistory</code> object. Another message in the <code>Message</code> table represents the clone of this message.		●	
SKIPPED	13	The administrator skipped this message.	●		

Some static properties on the `MessageStatus` class contains properties that contain arrays of message status values. You can use these to check values with easy to read code.

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), type the Gosu code:

```
gw.pl.messaging.MessageStatus.isAcked(Message.Status)
```

The following table lists additional `MessageStatus` static properties and the static methods.

MessageStatus static property or static method	Description
Properties	
ALL_STATES	An array of all states.
ACKED_STATES	An array of acknowledged states, including error states.
ACTIVE_STATES	An array of all active states.
BLOCKING_STATES	An array of active states for messages blocking sends of subsequent messages.
ERROR_STATES	An array of error states.
INACTIVE_STATES	An array of final message states for messages that no longer require processing.
PENDING_STATES	An array of all pending states.
RETRYABLE_STATES	An array of retryable states, PENDING_ACK or RETRYABLE_ERROR.
Methods	
isActive(state)	Returns true if the state is in the array ACTIVE_STATES.
isInFlight(state)	Returns true if the status indicates a message in flight (PENDING_ACK).
isRetryableError(state)	Returns true if the status is RETRYABLE_ERROR.
isRetryable(state)	Returns true if the status is in the array RETRYABLE_STATES
isPending(state)	Returns true if the status is in the array PENDING_STATES.
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.

Monitoring Messages and Handling Errors

PolicyCenter provides 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

Error Handling Concepts

Before describing the tools, it is useful to think about the kinds of errors that can occur and the actions that can be taken on these errors:

- **Pending send** – The message has not been sent yet:
 - If the message is related to an account, this is a safe-ordered message. The messaging destination may be waiting for an acknowledgement on the previous message for that same account.
 - The destination may be *suspended*, which means it is not processing messages
 - The destination may be simply not fast enough to keep up with how quickly the application generates messages. PolicyCenter can generate messages very quickly. For more information about how PolicyCenter retrieves messages from the send queue, see “Message Ordering and Multi-Threaded Sending” on page 323.

- **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 it is a *failure*, PolicyCenter suspends the destination automatically until an administrator restarts it. Failures include a retryable send error reaching its retry limit, or unexpected exceptions during the send method.

The destination also resumes (un-suspends) if the administrator removes the message or resynchronizes (resyncs) the message's primary object.

- **In-flight** – PolicyCenter waits for an acknowledgement for message it sent. If errors occur such that the external system does not receive or properly acknowledge 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. For more information, see “Safe Ordering” on page 292.

In this case, you can intervene to process the message (*skip* the unfinished message) or retry the message. 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 understand the actual status of the destination to make an informed decision about which correction to make.

- **Error** – The destination indicates that the message did not process successfully. Again, the error blocks sending subsequent messages. In some cases, the error message indicates that the error condition may 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.
- **Positively Acknowledged (ACK)** – The message successfully processed. These messages stay in the system until an administrator purges them. However, since the number of messages is likely to be very large, Guidewire recommends that you purge completed messages on a periodic basis.
- **Negatively Acknowledged (NAK)** – Message sending for that message failed at the external system (not a network error) either because the message had an error or was a duplicate.

If PolicyCenter retries a failed message, it marks the original message as failed/retried and creates a new copy of that message (with a new message ID) to send. The assumption behind this behavior is that a destination tracks messages received and does not accept duplicate messages. To do this, the retry must have a new message ID. If PolicyCenter retries an in-flight message because it never got an ACK, then it resends the original message 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 acknowledgement, then this prevents processing the message twice. The destination can send back an error (mark it as a duplicate) or send back another acknowledgement.

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 acknowledgement back to PolicyCenter.

PolicyCenter could become sufficiently out of sync with an external system such that simply skipping or retrying an individual message is insufficient to get both systems in sync. In such cases, you may need special administrative intervention and problem solving. Review your sever logs to determine the root cause of the problem.

PolicyCenter makes every attempt to avoid this problem. However, it provides a mechanism called resynchronizing (resyncing) to handle this case. All related pending and failed messages drop and resend. For more information, see “Resynchronizing Messages for a Primary Object” on page 339.

For more information about acknowledgements, see “Reporting Acknowledgements and Errors” on page 329.

The Administration User Interface

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.

Administrators have extensive errors across the system. In the Administration section of PolicyCenter, you can select the **Event Messages** console from the Administration page. There are three levels of detail for viewing events and messaging status:

- **Destinations List** – This shows a list of destinations, its sending status (for example, started or suspended), and counts of failed or in-process messages. At this level an administrator can only suspend or resume the entire destination. If suspended, PolicyCenter just holds all newly generated messages until the destination is resumes.
- **Destination Status** – This provides a list of policies that have failed messages or messages in-process for a single destination. Different filters can help you find different kinds of problems. You can also search for a particular policy and open the detail view for that policy. A user can then select one or more claims and indicate what to do with the failed or in-flight messages for each policy. Choose to skip, retry, or resync the policy.
- **Policy Details** – This shows a list of all failed or in-process messages for a policy (for all destinations). You can select messages that are in error or in-process and ask to skip or retry the messages.

Within the **Destination Status** screen, there is a special row for non-safe-ordered messages (cross-account messages) if any are in-process or failed. Most PolicyCenter messages relate to accounts, so all rows except that one row show problems related to accounts. Selecting the special row opens a list of all the non-safe-ordered messages.

From the **Administration** tab, you can force PolicyCenter to stop and restart the messaging sending queue without restarting the entire server. In rare situations, this is useful if you suspect that the sending queue became out of sync with messages waiting in the database. For example, perhaps your a network interruption in the cluster might cause such an issue. If the administrator restarts the messaging system, PolicyCenter shuts down each destination, then restarts the queue process, then reinitializes each destination.

Messaging tool actions such as **suspend**, **resume**, **retry**, and others can trigger from the Administration page and also from the **MessagingToolsAPI** web service. These messaging tools require the server’s run level to be **multiuser**.

Web Services for Handling Messaging Errors

In addition to monitoring and responding to errors with the administration user interface, PolicyCenter provides some other interfaces for dealing with errors. This section describes these tools.

First, PolicyCenter provides a web service called **MessagingToolsAPI**, which lets an external system remotely control the messaging system. See “**Messaging Tools Web Service**” on page 347.

Messaging Tools Web Service

PolicyCenter provides a web service called `MessagingToolsAPI`, which lets an external system remotely control the messaging system.

These API methods (except for `getPolicyMessageStatistics`) are available using the `messaging_tools` command line tool. See “Messaging Tools Command” on page 177 in the *System Administration Guide*. Within the administrative environment, the following command shows the syntax of each command:

```
messaging_tools -help
```

Retry a Message

To retry a message, call the `retryMessage` method. The behavior is the same as in the PolicyCenter user interface.

Skip a Message

To skip a message, call the `skipMessage` method. The behavior is the same as in the PolicyCenter user interface.

Retry Messages

To retry all *retryable* messages for a destination, call the `retryRetryableErrorMessages` method.

For example, call this method if the destination was temporarily unavailable and is now back on-line.

You can specify a maximum number of times to retry each message. This maximum prevents messages from retrying forever.

Purge Completed Messages

To purge completed messages, call the `purgeCompletedMessages` method. This method deletes completed messages in the messaging history table (`MessageHistory`) from the PolicyCenter database for all messages older than the given date.

Since the number and size of messages may be very large, Guidewire recommends you use this method periodically to purge old messages. Purging messages in the message history table prevents the database from growing unnecessarily large.

Always purge completed (inactive) messages before upgrading to a new version of PolicyCenter. Purging completed messages reduces the complexity of your upgrade.

Additionally, periodically use this command to purge old messages to avoid the database from growing unnecessarily.

Suspend Destination

To suspend a destination, call the `suspendDestination` method. Suspending a destination means that PolicyCenter stops sending messages to a destination.

PolicyCenter suspends the destination so that it can also release any resources such as a message batch file. Use this method to shut down the destination system to halt sending during processing of a daily batch file.

Resume Destination

To resume a destination, call the `resumeDestination` method. Resuming a destination means that PolicyCenter starts trying to send messages to the destination again.

If a previous suspend action released any resources, resuming the destination reclaims those resources. For example, the destination might reconnect to a message queue.

Get Messaging Statistics for a Safe Ordered Object

To get messaging statistics for a safe-ordered object, call the `getMessageStatisticsForSafeOrderedObject` object.

This method returns information that is similar to the user interface for a policy:

- the number of messages failed
- retryable messages
- in-flight messages
- unsent messages

Messaging tool actions such as `suspend`, `resume`, `retry`, and others can be triggered from the Administrator interface and also from the web services `MessagingToolsAPI` interface. These messaging tools can only be used if the server run mode is `multiuser`.

Change Messaging Destination Configuration Parameters

To change messaging destination configuration parameters on a running server, call the `configureDestination` method. This restarts the destination with the change to the configuration settings. The command waits for the destination to stop for the configured stop time. The method returns nothing,

The arguments are:

- `destID` – The destination ID of the destination to suspend
- `maxretries` – maximum retries
- `initialretryinterval` – initial retry interval
- `retrybackoffmultiplier` – additional retry backoff
- `pollinterval` – how often to poll, from start to start
- `numsenderthreads` – number of sender threads for multi-threaded sends
- `chunksize` – number of messages to read in a chunk
- `timeToWaitInSec` – the number of seconds to wait for the shutdown before forcing it

Get Configuration Information from a Destination

To get some configuration information from a messaging destination, call the `getConfiguration` method. This information is read from files on disk during server startup, however can be modified by web services and command line tools. See earlier in this topic for the `configureDestination` method.

The `getConfiguration` method takes only one argument, the destination ID. The method returns a `ExternalDestinationConfig` object, which contains properties matching the properties in the `getConfiguration` method, such as the polling interval and the chunk size. See the `getConfiguration` method documentation for details of each property.

Batch Mode Integration

Most integrations using the messaging system are real-time integrations between PolicyCenter and one or more destination systems. However, an external system might only be able to support batch updates. For example, some external server might need regular data from PolicyCenter and retrieves it only at night because it is resource intensive. In this case, PolicyCenter generates system events in real-time but sends updates in one batch to the external system.

There are essentially two approaches to handle batch messaging:

- **Approach 1: suspend a destination, then resume it later** – Using the SOAP API or command line tools, you can suspend sending messages to a destination during most of the day. If it is time to generate the batch file, you can resume (un-suspend) the destination so that PolicyCenter drains its queue of messages to generate the batch file. If there are no more messages (or after some period of time), you can suspend the destination again while you process the batch file or until a pre-defined time. This is a very safe method because it uses the messaging transaction and acknowledgement model to track each message.
- **Approach 2: append messages to a batch file and send all later** – PolicyCenter sends messages to a destination, which appends the messages to a batch file and immediately acknowledges the message. Periodically, the batch file is sent to the destination and processed. For example, after a certain number of messages are in the queue, then the message transport plugin can send them. Or, a separate process on the server can send these batch files. This approach allows you to send more than one message in a single remote call to the external system.

With both approaches, the message acknowledgement would come from the message transport plugin (or the message reply plugin) immediately, and not from the external system. With both approaches, this means that the messaging system built-in retry logic and ordered message sending cannot be used to deal with errors as gracefully as with a real-time integration. If any errors are found after sending an acknowledgement to PolicyCenter, your integration code must deal with these issues outside of PolicyCenter (or by issuing a resync request).

Included Messaging Transports

The Built-in Email Transport

PolicyCenter includes a built-in transport that can send standard SMTP emails. See “[Sending Emails](#)” on page 95 in the *Rules Guide* for details of configuration and APIs to send emails.

By default, the `emailMessageTransport` plugin use the *system user* to retrieve a document from the external system. You can chose to retrieve the document on behalf of the user who generated the email message. To do this, set the `a UseMessageCreatorAsUser` property in the `emailMessageTransport` plugin. In Studio, navigate to **Configuration** → **config** → **Plugins** → **registry** → `emailMessageTransport`. In the parameters area of the pane, click the **Add** button. Add the parameter `UseMessageCreatorAsUser` and set it to `true`.

Enabling the Built-in Console Transport

PolicyCenter includes a built-in console message transport example, which is an extremely simple messaging transport that writes the message text payload to the PolicyCenter console window. Enable this transport in the plugin registry in Studio to debug integration code that creates and sends messages. For more detail of how to do this, see the *Studio Guide*.

If you register a messaging plugin, you must register it in two places. First, register it in the plugin registry in the plugin editor; see “[Using the Plugins Registry Editor](#)” on page 131 in the *Configuration Guide*. Next, register it in the messaging registry in the messaging editor; see “[Using the Messaging Editor](#)” on page 153 in the *Configuration Guide*.

In the plugin editor, use the plugin name `consoleTransport`, the interface name `MessageTransport`, the Java class `examples.plugins.messaging.ConsoleMessageTransport` and the plugin directory `messaging`.

In the messaging editor, use the following settings:

- set the plugin name to "Console Message Logger"
- set transport plugin name to `consoleTransport`
- set initial retry interval set to 100
- set max retries set to 3
- set `retrybackoffmultiplier` to 2

- set event name to "\w*", which means the destination wants notification of all events

This tells PolicyCenter to send all events to this destination, and trigger Event Fired rules accordingly. write Event Fired rules that create messages for this destination

After redeploying the server, watch the console window for messages.

part V

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. For an overview of PolicyCenter rating integration, see “The Rating Framework” on page 353. In particular, “Where to Override the Default Rating Engine?” on page 360 discusses various rating integration strategies you can use.

This topic describes the relationship between cost entity instances and cost data objects, the rating plugin, and how to implement a line-specific rating engine class. However, this topic does not describe details of the *Cost* and *Transaction* subtypes specific to each line of business. For that information, see “Entities Associated with Costs and Transactions” on page 428 in the *Application Guide*. Also refer to Data Dictionary for details of the root cost entity for each line of business. For example, *PACost* is the root cost entity for personal auto costs.

This topic includes:

- “The Rating Framework” on page 353
- “Implementing Rating for a New Line of Business” on page 367
- “Rating Line Example for Personal Auto” on page 393
- “Rating Variations” on page 397

See also

- For detailed information on the terms *slice mode*, *window mode*, and *version lists* as used in this topic, see:
 - “Policy Revisioning” on page 489 in the *Application Guide*
 - “Structure of Revisioning Across Effective Time” on page 497 in the *Application Guide*

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 (step 4 in the previous numbered list). This topic discusses how to integrate your own rating code into PolicyCenter. For a general overview of the quoting process in the user interface, see “Quoting and Rating” on page 427 in the *Application Guide*.

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.

Note: The rest of this topic uses terminology of the PolicyCenter policy revisioning system. For important discussion about these concepts, see “Policy Revisioning” on page 489 in the *Application Guide*. In particular, see the topic “Structure of Revisioning Across Effective Time” on page 497 in the *Application Guide*.

When PolicyCenter needs to rate a policy, it calls the registered implementation of the rating plugin interface (`IRatingPlugin`) 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 a different built-in plugin implementation class. For details, see “Guidewire Rating Management and PCRatingPlugin” on page 357.

IMPORTANT To determine whether your Guidewire PolicyCenter license agreement includes Guidewire Rating Management, contact your Guidewire sales representative. Rating Management requires an additional license key. For instructions on obtaining and installing this key, contact your Guidewire support representative.

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.

Note: 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.

WARNING 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 carriers are in separate rating engine classes for each line of business. Each line-specific rating engine extends the abstract rating engine class (`AbstractRatingEngine`). Additionally, consider encapsulating your actual algorithm into separate Gosu classes, one for each type of premium. For how to write your rating algorithm, including table lookups, see “Coding Your Actual Rating Algorithm” on page 390.

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 is the following:

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.

Note: For information about slice mode and related revisioning concepts, refer to “Structure of Revisioning Across Effective Time” on page 497 in the *Application Guide*.

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. For a reference of cost properties, including which ones are required to set, see “Cost Core Properties” on page 368. 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. To implement asynchronous rating instead, see “Optional Asynchronous Rating” on page 366.

Note: 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. For how to write your rating algorithm, including table lookups, see “Coding Your Actual Rating Algorithm” on page 390.

4. **Combine costs for all slices and prorate the costs** – PolicyCenter merges any costs if two costs match all of the following:

- 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. For more about the proration algorithm, see “Is Proration Always Linear?” on page 364.

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. For more information about flat-rated costs, see “Cost Delegate” on page 429 in the *Application Guide*.

Note: 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. For how to write your rating algorithm, including table lookups, see “Coding Your Actual Rating Algorithm” on page 390.

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 *cost data objects* (`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. For more information, see “Overview of Cost Data Objects” on page 358. 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. For more information about key values, see “Key Values for Each Cost Data Subclass” on page 377. 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. For alternative rating strategies, see “Rating Variations” on page 397.

See also

- For how the default rating engine uses cost data objects, see “Overview of Cost Data Objects” on page 358.
- For where to add your main rating code, see “Where to Override the Default Rating Engine?” on page 360.

- For a feature-level overview of quoting, see “Calculating Transactions” on page 442 in the *Application Guide*.
- For data model entities for each line of business, see “Entities Associated with Costs and Transactions” on page 428 in the *Application Guide*.

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 all other lines of business, the plugin calls its superclass `SysTableRatingPlugin` to create the default rating engine instances.

For more information, see “Rating Management” on page 541 in the *Application Guide*.

To 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:
 - Active
 - Approved
 - Stage
 - Draft
- For more information about this parameter, see “Minimum Rating Level Parameter” on page 548 in the *Configuration Guide*.
4. Obtain and install a license key for Guidewire Rating Management.

IMPORTANT To determine whether your Guidewire PolicyCenter license agreement includes Guidewire Rating Management, contact your Guidewire sales representative. Rating Management requires an additional license key. For instructions on obtaining and installing this key, contact your Guidewire support representative.

Extending Rating to Other Lines of Business

To use Rating Management with lines of business other than commercial property and personal auto:

1. 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.
2. 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.

For more details of both these steps, see “Checklist for Creating a New Policy Line Rating Engine” on page 385.

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.

For the structure of the cost entities for each built-in line of business, refer to the following topics:

- “Cost and Transaction Model for Personal Auto Line” on page 439 in the *Application Guide*
- “Cost and Transaction Model for Businessowners Line” on page 432 in the *Application Guide*
- “Cost and Transaction Model for Workers’ Compensation Line” on page 441 in the *Application Guide*

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.

IMPORTANT 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. For alternate strategies, see “Where to Override the Default Rating Engine?” on page 360.

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 <i>fixed ID</i> 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. For more information about Fixed IDs, see “Structure of Revisioning Across Effective Time” on page 497 in the <i>Application Guide</i> . 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. See “Fixed ID Keys Link a Cost Data Object to Another Object” on page 375.
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.
How easy is it to write your own code to split or merge costs across effective time?	It is difficult to split or merge <code>Cost</code> 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. For more information about window mode, see “Structure of Revisioning Across Effective Time” on page 497 in the <i>Application Guide</i> .	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 <code>Cost</code> entities based on your <code>CostData</code> 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. See “Cost Data Object Methods and Constructors to Override” on page 376.
- **API methods that you can call** – Every cost data class inherits built-in APIs from the `CostData` class. See “Cost Data APIs That You Can Call” on page 382.

See also

- “What Do Cost Data Objects Contain?” on page 368

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.

Note: These steps are equivalent to the higher-level list of steps the beginning of “The Rating Framework” on page 353. See that topic for expanded feature-level discussion.

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.

Note: For conceptual information about slice mode, refer to “Structure of Revisioning Across Effective Time” on page 497 in the *Application Guide*.

- 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.

Note: 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. For how to write your rating algorithm, including table lookups, see “Coding Your Actual Rating Algorithm” on page 390.

- c. Merge the cost data objects from all slices where possible.
- d. Calculate prorated amounts. This step assumes *linear proration*. If you think this assumption does not apply to you, see “Is Proration Always Linear?” on page 364. 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.

Note: 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. For how to write your rating algorithm, including table lookups, see "Coding Your Actual Rating Algorithm" on page 390.

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 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. (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	<p>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.</p> <p>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. For more details, see "Workers' Compensation Rating" on page 398.</p>
Do everything on your own with raw cost entities	The entire rating plugin.	<p>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.</p>

Plugin for Guidewire Rating Management

If you use Guidewire Rating Management, you use a different built-in plugin implementation class than the default rating plugin. For details, see “Guidewire Rating Management and PCRatingPlugin” on page 357.

IMPORTANT To determine whether your Guidewire PolicyCenter license agreement includes Guidewire Rating Management, contact your Guidewire sales representative. Rating Management requires an additional license key. For instructions on obtaining and installing this key, contact your Guidewire support representative.

Common Questions About the Default Rating Engine

The following are common rating strategy questions:

- What are Rate-scalable and Basis-scalable Costs?
- Can We Always Rate Only From the Current Slice Forward?
- Can We Rate Only If Something Important Changed?
- Can We Rate the Whole Policy for Each Slice, Rather Than One Line at a Time?
- Is Proration Always Linear?
- How Does Proration Handle Minor Differences in Term Length?

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 engine looks up a *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 looked up from a table. Next, a rating engine modifies that rate based on a variety of other factors (vehicle cost, driver age, and so on) 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*. For more information, see “Specifying Whether to Merge a Cost as Basis Scalable” on page 381.

See also

- For more information about the cost terms *base rate*, *adjusted rate*, *term amount*, *amount*, and *number of rated days*, see the table in “Cost Core Properties” on page 368.
- For more information about proration calculations, see “Is Proration Always Linear?” on page 364.

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*. See “Out-of-sequence Jobs” on page 511 in the *Application Guide*.

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. See “Decide Whether to Rate Only From Change Date Forward” on page 389 for details of this flag and the default behavior.

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:

- 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.

WARNING 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 One Line at a Time?

Some customers ask if when rating a multi-line policy, they can rate one slice at a time across all lines rather than rating each line independently. The reason to do this might be because the slice calculation might include discounts that depend on premiums across the entire policy. The default rating approach does not automatically support this approach.

With additional customization, you could do this if necessary but requires additional work:

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 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 step 2.
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. See “Cost Core Properties” on page 368 for more about this and other properties.

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)
```

Note: For more about these cost properties such as `Cost.TermAmount` and `Cost.NumDaysInRatedTerm`, see “Cost Core Properties” on page 368

For most customers in 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 default rating engine.

If you want to change this logic, refer to the class `gw.plugin.policyperiod.impl.SysTableRatingPlugin` 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. For more details, see “Where to Override the Default Rating Engine?” on page 360. 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.

See also

- For a related discussion regarding the number of days in the term calculation, see “How Does Proration Handle Minor Differences in Term Length?” on page 365.

How Does Proration Handle Minor Differences in Term Length?

As discussed in “Is Proration Always Linear?” on page 364, 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. See “Cost Core Properties” on page 368 for more about this and other properties.

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 do the following:

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 formula:
$$\text{period_rate} = \text{actual_rate} * (\text{number_of_days_in_period}) / (\text{number_of_days_for_standard_rate})$$
.
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. If you have questions about this, contact Guidewire Customer Support.

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.

IMPORTANT Guidewire recommends that you write your rating code to rate policies synchronously.

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.

To enable asynchronous rating

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. See “Where to Override the Default Rating Engine?” on page 360 for a comparison of rating integration strategies.

IMPORTANT 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.
For more information, see “Messaging and Events” on page 289 and specifically “Triggering Custom Event Names” on page 316.
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.
For more information, see “Messaging and Events” on page 289 and specifically “Generating New Messages in Event Fired Rules” on page 317.
6. Write a messaging transport that calls out to your rating external system.
For more information, see “Messaging and Events” on page 289 and specifically “Message Destination Overview” on page 302.
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.

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.

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`.

As described in more detail in “Where to Override the Default Rating Engine?” on page 360, 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:

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 (the basis typically is 1)
4. Set the *amount* to the value of the 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.

For more information about rating integration strategy possibilities, see “Where to Override the Default Rating Engine?” on page 360. 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 390.

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. For a list of important cost-related properties common to all costs, see “Cost Core Properties” on page 368. 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. For details, see “Cost Data Object Methods and Constructors to Override” on page 376.
- **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. For details, see “Cost Data Object Methods and Constructors to Override” on page 376.
- **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. For details, see “Cost Data APIs That You Can Call” on page 382.

Cost Core Properties

The `Cost` entity is the core output of the rating engine. As mentioned in more detail in “The Rating Framework” on page 353, 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. Users can override some of the cost information, see “Configuring Rating Overrides” on page 458 in the *Application Guide*. 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	Basis	<p>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.</p> <p>For owned property, typically the basis is 1, which represents one car or one building, for example.</p> <p>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.</p> <p>In your rating code, always explicitly set the <code>Basis</code> property.</p>
Base rate	ActualBaseRate StandardBaseRate OverrideBaseRate	<p>The cost rate percentage prior to applying any discounts or adjustments. The base rate typically comes from a rate table.</p> <p>In your rating code, set the property explicitly as follows:</p> <ul style="list-style-type: none"> • <code>ActualBaseRate</code> – Always set explicitly. If there are overrides, the rating engine might set the <code>actual</code> 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 <code>override</code> property to <code>null</code> if there are no overrides.
Adjusted rate	ActualAdjRate StandardAdjRate OverrideAdjRate	<p>The cost rate percentage after applying modifier factors such as discounts and adjustments.</p> <p>In your rating code, set the property explicitly as follows:</p> <ul style="list-style-type: none"> • <code>ActualAdjRate</code> – Always set explicitly. If there are overrides, the rating engine might set the <code>actual</code> 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 <code>override</code> property to <code>null</code> if there are no overrides.
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 <i>for the entire term</i>. The term length is the value in the property <code>NumDaysInRatedTerm</code>. For the prorated value (using effective and expiration dates), see the row with label "amount".</p> <p>In your rating code, set the property explicitly as follows:</p> <ul style="list-style-type: none"> • <code>ActualTermAmount</code> – Always set explicitly. • <code>OverrideTermAmount</code> – Set only if the rating line supports overrides. • <code>StandardTermAmount</code> – 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 <code>Cost</code> entity.</p> <p>In your rating code, always explicitly set the <code>EffectiveDate</code> property.</p>

Cost information	Property names on costs and cost data objects	Description
Proration method	ProrationMethod	<p>The proration method. The built-in choices are:</p> <ul style="list-style-type: none"> • ProRataByDays – A prorated cost (the default) • Flat – A flat cost <p>For more information about flat costs, see “Cost Delegate” on page 429 in the <i>Application Guide</i>.</p> <p>With Guidewire Rating Management, you can define and rate flat costs.</p> <p>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</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 <i>term amount</i> 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>

Cost information	Property names on costs and cost data objects	Description
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). For more information about pro rata calculations, see “Is Proration Always Linear?” on page 364.</p>
		<p>In your rating code, set the property explicitly as follows:</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, which means, the ActualAmount was overridden. An external rating engine might set ActualAmount explicitly to prevent merging and proration for some reason. • 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 gw.rating.CostData class.</p>
		<p>IMPORTANT: The ActualAmount property is very important. That 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:</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 <i>payroll reporting</i> 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>

Cost information	Property names on costs and cost data objects	Description
Overridable	Overridable	<p>Can this cost be overridden by editing this cost in the Premium Overrides screen? If this property is <code>false</code>, 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 <code>true</code>.</p>
		<p>Explicitly set the <code>Overridable</code> property if you do not want the default value (<code>true</code>). 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 (<code>RateAmountType</code>) property.</p>
		<p>In your rating code, always explicitly set the <code>ChargePattern</code> property.</p>
Charge group	ChargeGroup	<p>Another optional subcategorization for costs.</p>
		<p>In your rating code, always explicitly set the <code>ChargeGroup</code> property.</p>
Override reason	OverrideReason	<p>An explanation for why the user overrode this cost. The value is descriptive only. PolicyCenter does not use it to calculate premiums.</p>
		<p>Explicitly set the <code>overrideReason</code> property only if the rating line supports overrides.</p>
Merge as basis scalable	MergeAsBasisScalable	<p>Specifies whether during cost merging, whether to prorate the basis value based on the time length.</p>
		<p>Never explicitly set the <code>MergeAsBasisScalable</code> property. The property is read-only but non-final. Only override this property in CostData subtypes for basis-scalable costs. By default, it returns <code>false</code>.</p>
Cost key	Key	<p>An internal ID property that PolicyCenter creates on cost objects. Not directly mirrored on cost data objects.</p>
		<p>Never explicitly set the <code>Key</code> property. The property is read-only and final. You cannot override it or set it.</p>
Intrinsic type	IntrinsicType	<p>An internal ID property that PolicyCenter creates on cost objects. Not directly mirrored on cost data objects.</p>
		<p>Never explicitly set the <code>IntrinsicType</code> property. The property is read-only and final. You cannot override it or set it.</p>

Note: For related discussion of setting these properties in your rating engine, see the related topics. See “Implementing Rating for a New Line of Business” on page 367 and “Coding Your Actual Rating Algorithm” on page 390.

Special Notes about Overrides

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.

For related information, see “Handling Premium Overrides” on page 391.

Special Notes about 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.

For example, this code 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.FixedId)
    populateCostData(cost, baseRate, adjRate)
    return cost
}

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()
}
```

Adding Line-specific Cost Properties and Methods

Remember that each line of business defines its own root cost entity. Every root `Cost` entity must implement the `CostAdapter` interface and delegate it to a separate class that you create. This *cost adapter* interface has a single method on it 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`) would implement this interface, so typically you do not need to reimplement it in any subtypes.

For example:

```
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
    }
}
```

Additionally, most line-specific costs (subtypes of the root cost entity) must 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.

The code looks like the following:

```
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.

For a list of cost-related properties that are common to all costs, see “Cost Core Properties” on page 368. Additionally, a cost data object might contain:

- Properties in common to all costs in the line. See “Adding Line-specific Cost Properties and Methods” on page 374.
- 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. For more information about fixed IDs, see “Structure of Revisioning Across Effective Time” on page 497 in the *Application Guide*.

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
//...
}

```

Note: In Gosu, class variables without an access modifier like `private` or `public` default to `private`.

To get the fixed ID Key object from an entity, simply access its `FixedID` property. For example:

```
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. See “Copying Custom Properties from Cost Data Objects to Cost Entity Instances” on page 378 for important related discussion.

IMPORTANT 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. See “Structure of Revisioning Across Effective Time” on page 497 in the *Application Guide* for more revisioning details.

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 (implement) certain 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:

- If the rating engine determines 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.

IMPORTANT Guidewire strongly recommends that the constructor includes any subtype-specific properties in the constructor rather than setting these properties after instantiation.

For example, from the personal auto cost data subtype `PersonalAutoCovCostData`:

```
// constructor that takes the effective and expiration dates. It sets local variables as needed.
construct(effDate : Date, expDate : Date, vehicleIDArg : Key, covIDArg : Key) {
    super(effDate, expDate)
    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 immediately prior to the effective date of the change.
 2. The rating engine uses the cost data constructor that takes an existing cost.
 3. The constructor creates a cost data object that matches the existing cost.

To support this, 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.

For example, from the personal auto cost data subtype `PersonalAutoCovCostData`:

```
// constructor that takes the specific type of Cost entity. It sets local variables as needed.
construct(c : PersonalAutoCovCost) {
    super(c)
    _vehicleID = c.PersonalVehicle.FixedID
    _covID = c.PersonalAutoCov.FixedID
}
```

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 mergeable if they satisfy all of the following:

- 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.

Note: 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 mergeable 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.

Note: 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. See "Fixed ID Keys Link a Cost Data Object to Another Object" on page 375.

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:

```
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.

WARNING Carefully consider how you implement the `KeyValues` method. Errors in this method corrupt rating data during the merge process. If you have questions about which properties to include in your `KeyValues` method, contact Guidewire Customer Support.

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.

Note: 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 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`.

In contrast, for cost data objects, the key has type `CostDataKey` and includes:

- The `CostData` subtype
- The `ChargePattern`, `ChargeGroup`, and `RateAmountType` properties
- Any values in the `CostData.KeyValues` property for that cost data subclass.

Note: See earlier in this topic for details writing `property get` function for the `KeyValues` property in the cost data object subtype.

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 Cost 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()`.

Note: 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:

```
// 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 property name on the *Cost* entity, as a *String*.
- 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:

```
override function setSpecificFieldsOnCost(line : PersonalAutoLine, cost : PersonalAutoCovCost) : void {  
    super.setSpecificFieldsOnCost(line, cost)  
    cost.setFieldValue( "PersonalAutoCov", _covID )  
    cost.setFieldValue( "PersonalVehicle", _vehicleID )  
}
```

Get 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. See “Key Values for Each Cost Data Subclass” on page 377.

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.

Note: Before starting to write this code or reading the rest of this topic, consider reviewing the revisioning documentation. For critical conceptual information about how PolicyCenter stores objects that change across a policy term, see “Structure of Revisioning Across Effective Time” on page 497 in the *Application Guide*. For important information about version lists and version list APIs, see “Version List API Reference” on page 504 in the *Application Guide*.

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 1 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. See “Structure of Revisioning Across Effective Time” on page 497 in the *Application Guide* and “Fixed ID Keys Link a Cost Data Object to Another Object” on page 375.

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`. Its method signature is:

```
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 follows:

```
// 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.

Note: 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. For more about version list generated properties, see “Window Mode API Overview” on page 501 in the *Application Guide*.

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.

Note: 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. For important conceptual information about version lists, see “Structure of Revisioning Across Effective Time” on page 497 in the *Application Guide*.

The code for this method looks like the following:

```
override function getVersionedCosts(line : PersonalAutoLine) :  
    List<com.guidewire.commons.entity.effdate.EffDatedVersionList> {  
        var covVL = EffDatedUtil.createVersionList( Line.Branch, _covID ) as PersonalAutoCovVersionList  
        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  
}
```

In this example, it chooses the coverage version with earliest effective date. This might seem arbitrary, and in some sense it is. However, cost cannot represents 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 and dereference its vehicle and get the fixed ID from that.

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, for example:

```
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.

WARNING You can call the following API methods, but do not attempt to modify or override them.

Debugging

For debugging use, it may be helpful to use the cost data object method `debugString`. It generates a debug string containing the core rate/amount/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`.

Note: Internally, this method calls the `getVersionedCosts` method, which each cost data must override.
See “Get Versioned Costs That Match This Cost Data Object” on page 379.

For example:

```
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.

Note: Unlike the `getExistingCost` method, the `getOrCreateCost` method never returns `null`.

For example:

```
var c = getOrCreateCost(myPolicyLine)
```

Finally, there is another method called `getPopulatedCost`. It does the following:

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. See “Cost Core Properties” on page 368.
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. See “Cost Core Properties” on page 368.

For example:

```
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. See “Fixed ID Keys Link a Cost Data Object to Another Object” on page 375 for important details.

To add another relationship (foreign key) to a cost data subclass, do 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`. See “Fixed ID Keys Link a Cost Data Object to Another Object” on page 375 for related discussion.
2. Add new arguments to the constructor for these new values. In the constructor, set your private variables based on the constructor arguments. See “Constructors for a Cost Data Subclass” on page 376 for related discussion.
3. Override the cost data method `setSpecificFieldsOnCost`. Set your foreign key properties when creating a new cost. See “Copying Custom Properties from Cost Data Objects to Cost Entity Instances” on page 378 for related discussion.

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. See “Key Values for Each Cost Data Subclass” on page 377 for related discussion. 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.

Note: 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`. See “Get Versioned Costs That Match This Cost Data Object” on page 379 for related discussion. 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.

Note: 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 something that is:

- 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. The `rateOnly` method encodes the default algorithm described in “The Rating Framework” on page 353. 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.

Note: For a typical rating integration project, implementing your actual rating algorithm, typically in `rateSlice` and `rateWindow`, is the majority of your engineering time.

For detailed discussion on potential rating strategies, see “Where to Override the Default Rating Engine?” on page 360. You might want to read that topic before starting any rating integration programming.

Checklist for Creating a New Policy Line Rating Engine

The following list summarizes the required steps to create a rating engine for a new typical policy line:

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. For example for the `CommercialPropertyLine`:

```
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_SYSTABLE`:

```
override function createRatingEngine(method : RateMethod,  
    parameters : Map<RateEngineParameter, Object>) : AbstractRatingEngine<PersonalAutoLine> {  
    if (RateMethod.TC_SYSTABLE == method) {  
        return new PASysTableRatingEngine(_line as PersonalAutoLine)  
    }  
    return new PARatingEngine(_line as PersonalAutoLine,  
        parameters[RateEngineParameter.TC_RATEBOOKSTATUS] as RateBookStatus)  
}
```

Note: For more information about Guidewire Rating Management and the `PCRatingPlugin`, see “Guidewire Rating Management and `PCRatingPlugin`” on page 357.

3. Create a `CostData` subclass for the line and further subclasses of `CostData` for each subclass of cost on the line. See “Overview of Cost Data Objects” on page 358 and the first few topics in “Implementing Rating for a New Line of Business” on page 367.

4. Decide whether your line can be rated from the change date forward only or whether PolicyCenter must re-rate the whole period. See “Decide Whether to Rate Only From Change Date Forward” on page 389.

5. Override methods on `AbstractRatingEngine` to actually do the rate calculations. The exact methods might vary. For detailed discussion on potential rating strategies, see “Where to Override the Default Rating Engine?” on page 360. You might want to read that topic before starting any rating integration programming.

6. Consider encapsulating the logic for each type of rating in a separate classes for each type of rating algorithm. This is where you can put your actual rating formulas and table lookups. Wherever your actual rating code lives, separate your table lookup code from your formulas.

Note: For more information how to implement table lookups and how to handle changes to your rating algorithm, see “Coding Your Actual Rating Algorithm” on page 390.

7. If this line handles premium overrides, add relevant logic. See “Handling Premium Overrides” on page 391.

For more information about rating integration strategy possibilities, see “Where to Override the Default Rating Engine?” on page 360. 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 390.

For a step-by-step description of implementing a rating engine for real line of business, see “Rating Line Example for Personal Auto” on page 393.

Additionally, review the Gosu source code for other rating engines for the approaches that other built-in lines of business use. Note that workers' compensation and general liability rating work very differently from typical lines of business. For an overview of rating in unusual lines of business, see "Rating Variations" on page 397.

Methods and Properties of the Abstract Rating Engine Class

The following table summarizes the methods in the abstract rating class to override or use. The rightmost column indicates whether to expect to override this method in a new rating engine. The methods that appear earliest in the table are methods that a new rating engine would typically override. Some methods listed later are helper methods for the default rating engine, or internal methods that you do not need to override unless you significantly change the logic. Utility methods perform functions such as the following:

- Merging and prorating CostData objects
- Calculating the number of days in the rated term based on the default policy term for the product
- Calculating a demo tax rate
- Calculating a short rate penalty rate

AbstractRatingEngine method (or property as noted)	Description	Override it?
existingSliceModeCosts	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 false, this method is unused.
rateSlice	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. See "Where to Override the Default Rating Engine?" on page 360. 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 390.	Yes, for typical rating lines. If you completely override the <code>rateOnly</code> method, this method is unused.
rateWindow	Rates the policy in <i>window mode</i> . This is where you would create costs that either: <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. The argument is the <i>version</i> of the policy line that is earliest in effective time. See "Where to Override the Default Rating Engine?" on page 360. 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 390.	Yes, for typical rating lines. If you completely override the <code>rateOnly</code> method, this method is unused.
createCostDataForCost	Given the specified <code>Cost</code> entity instance, creates the appropriate <code>CostData</code> subclass and returns it. The <code>extractCostDatasFromExistingCosts</code> method calls this class. The implementation of the <code>createCostDataForCost</code> method must prepare for any <code>Cost</code> 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 false, this method is unused.

AbstractRatingEngine method (or property as noted)	Description	Override it?
rateOnly	<p>The core rating loop, with the following actions:</p> <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. For related discussion, see “Decide Whether to Rate Only From Change Date Forward” on page 389 and “Can We Always Rate Only From the Current Slice Forward?” on page 362. • Merge cost data objects. • Prorate cost data objects. • Rate window costs. <p>See “Where to Override the Default Rating Engine?” on page 360. 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 390.</p>	Only if you want to bypass the general logic of the rating engine.
shouldRateThisSliceForward	<p>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. For more information, see “Decide Whether to Rate Only From Change Date Forward” on page 389 and “Can We Always Rate Only From the Current Slice Forward?” on page 362</p>	Only if you want to change the default logic. See the topics mentioned earlier in this table row.
ShortRatePenaltyRate property	<p>This property get 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 (WCRatingContext). The default rating engine always returns 10% but you can override this.</p>	Yes
getStateTaxRate	<p>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.</p>	Yes
NumDaysInCoverageRatedTerm property	<p>This property get 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. For default purposes, the built-in version determines this on the default term type for the product and only supports a couple built-in term types. In a real-world production environment, you would probably hard-code this value into the rating engine. Alternatively, you might use this number to determine which rate tables to use. Also see “How Does Proration Handle Minor Differences in Term Length?” on page 365</p>	Yes
rate	<p>This is the entry point for the rating request. The rating plugin calls this method for each rating line. It does the following:</p> <ul style="list-style-type: none"> • Calls the rateOnly method on this object, which generates cost data objects. • Calls the attachCostEntities method to convert the cost data objects to actual Cost entities • As part of the attachCostEntities method, removes any old Cost entities for this policy line that were untouched by the rateOnly method. 	No. Generally speaking, do not override this. For typical rating lines, just override rateSlice and rateWindow. If you must bypass or rewrite the built-in rating control structure, generally speaking, override the rateOnly method, not the rate method.
assertSliceMode	<p>Asserts that the specified revised entity instance is in slice mode. This is a general utility function you can use.</p>	No. It is a general utility function you can use.
assertWindowMode	<p>Asserts that the specified revised entity instance is in window mode.</p>	No. It is a general utility function you can use.

AbstractRatingEngine method (or property as noted)	Description	Override it?
getNextSliceDateAfter	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.	No
	This useful to set the effective and expiration dates of the cost. See "Writing Your Own Line-specific Rating Engine Subclass" on page 384	
extractCostDatasFromExistingCosts	<p>This method take a list of costs and a cut-off date and does 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 ActualAmount 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>. For related discussion, see "Decide Whether to Rate Only From Change Date Forward" on page 389 and "Can We Always Rate Only From the Current Slice Forward?" on page 362.</p> <p>You probably do not need to use this API unless you greatly change the logic of the class.</p>	No
addCost	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
prorateToCutOffDate	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.	No
attachCostEntities	<ul style="list-style-type: none"> • Converts the cost data objects to actual Cost entities • Removes any old Cost entities for this policy line that were untouched by the <code>rateOnly</code> method. 	No
validate	Ensures that for any given period of time, there is only one Cost entity instance with a given cost key (<code>CostKey</code>). 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
mergeCosts	Merge any equal costs that are <i>attributed to the same elements</i> (they have matching <code>cost keys</code>) and adjacent in effective time. This method returns a new list of costs rather than modifying the existing list of costs in place.	No
attemptToMerge	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
updateAmount	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.	No

Decide 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. For related overview discussion, see “Can We Always Rate Only From the Current Slice Forward?” on page 362.

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.

The behavior of the built-in implementation is as follows:

- 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. For the reference for all cost data object properties, see the table in “Cost Core Properties” on page 368. Carefully read the rightmost column in that table to see what properties are required to set, and in which cases.

Cloning a Cost into a New Cost Data Object

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

For example, from the personal auto rating engine:

```
override function createCostDataForCost(c : Cost) : CostData {  
    switch (typeof c) {  
        case PersonalAutoCovCost: return new PersonalAutoCovCostData(c)  
        case PersonalVehicleCovCost: return new PersonalVehicleCovCostData(c)  
        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 carrier business users manage
- Programming logic in rating formulas, which typically IT departments own and manage

Note: 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.

IMPORTANT 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:

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. See “Query Builder APIs” on page 129 in the *Gosu Reference Guide*. 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.

Thus, all of the following are important rules that you must follow:

- **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 look like the following:
 - a. Check the effective date of the change.
 - b. If the change's effective date is before the formula's change date, your code must run your old formula.

WARNING It is critical that you never permit backdated rates in production systems.

- c. 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.

WARNING 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.

For more information about where to actually inject your rating code, see “Where to Override the Default Rating Engine?” on page 360.

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:

- 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. See “Getting Cost Entity Instances for This Cost Data Object” on page 382.

- 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`. See the table in “Cost Core Properties” on page 368. 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`.

WARNING 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.

Note: The built-in workers’ compensation rating line shows how different override properties to adjust from using standard values to using overridden values. See “Workers’ Compensation Rating” on page 398 for more information.

To add support for premium overrides

1. Check if there is an existing cost.
2. If so, check if there are non-null values for any override properties. For a detailed reference for override-related cost properties, see “Cost Core Properties” on page 368.
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 typically calculates that there is zero cost (\$0) for a particular cost in some cases. Normally, this means the rating engine does not bother to generate a cost row. That choice is problematic because the premium overrides user interface allows you to override existing rows but not add new Cost rows. In other words, 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. However, to edit them 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, 2010 to January 1, 2011 and the audit period is for February 1, 2010 through March 1, 2010. Rating must not return any costs that are effective prior to February 1, 2010 or which expire later than March 1, 2010. To prevent mistakes, PolicyCenter detects this condition and throws an exception for any rule violations.

The cost (and cost data) property `SubjectToReporting` property controls what happens to the rating and billing of certain types of costs for policies subject to reporting. This 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, in 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.

Rating Line Example for Personal Auto

This topic describes the implementation of one of the built-in lines of business, Personal Auto (PA). Although all lines of business are different, use this information to understand the types of things 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. See “The Rating Framework” on page 353.

Note: For a summary of necessary steps when writing rating for a new line, see “Checklist for Creating a New Policy Line Rating Engine” on page 385.

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 details, see “Guidewire Rating Management and PCRatingPlugin” on page 357.

IMPORTANT To determine whether your Guidewire PolicyCenter license agreement includes Guidewire Rating Management, contact your Guidewire sales representative. Rating Management requires an additional license key. For instructions on obtaining and installing this key, contact your Guidewire support representative.

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>PersonalAutoTaxCost</code>	A cost value for taxes	<code>PersonalAutoTaxCostData</code>
<code>PAShortRatePenaltyCost</code>	A cost value that represents a cancellation short rate penalty.	<code>PAShortRatePenaltyCostData</code>

Personal auto cost entity	Description	Personal auto cost data class
PAMultiPolicyDiscCost	A cost value that represents multipolicy discounts	PAMultiPolicyDiscCostData
PersonalVehicleCovCost	A cost value for a vehicle coverage that has a link to PersonalVehicleCov. The link to the coverage already implies a link to the vehicle.	PersonalVehicleCovCostData
PersonalAutoCovCost	A cost value that adds a link to a PersonalVehicleCov and a link to a PersonalVehicle. 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.	PersonalAutoCovCostData

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.

See also

- For more about fixed ID Key objects, see:
 - “What Do Cost Data Objects Contain?” on page 368
 - “Fixed ID Keys Link a Cost Data Object to Another Object” on page 375

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. It extends the PACostData class, parameterized on the corresponding entity type `PersonalAutoCovCost`. The class declares its superclass in Gosu generics syntax as follows:

```
class PersonalAutoCovCostData extends PACostData<PersonalAutoCovCost> {
```

The angle bracket notation indicates use of Gosu generics. This particular example means that we want the PACostData generic class to work with the class `PersonalAutoCovCost`. This allows the definition of PACostData to be written in a general way to work with lots of types of data, but be typesafe at compile time. Some of the rating code uses this parameter to determine which Cost entity subtype to create. For more information about generics, see “Gosu Generics” on page 243 in the *Gosu Reference Guide*.

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
```

Note: For more about fixed ID Key objects, see “What Do Cost Data Objects Contain?” on page 368 and “Fixed ID Keys Link a Cost Data Object to Another Object” on page 375.

Constructors for PersonalAutoCovCostData

The standard constructor initializes these properties. The constructor also ensures that the fixed ID Key objects passed to the constructor have the proper type. This avoids errors like switching the order of the arguments:

```
construct(effDate : Date, expDate : Date, vehicleIDArg : Key, covIDArg : Key) {
  super(effDate, expDate)
  assertKeyType(vehicleIDArg, PersonalVehicle)
  assertKeyType(covIDArg, PersonalAutoCov)
  _vehicleID = vehicleIDArg
  _covID = covIDArg
}
```

There is an alternate 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
}

```

For more information about cost data constructors, see “Constructors for a Cost Data Subclass” on page 376

Set Specific Properties on Cost for PersonalAutoCovCostData

The cost data `setSpecificFieldsOnCost` method needs to populate the line-specific (non-core) properties. For important details about this method and how to implement it, see “Copying Custom Properties from Cost Data Objects to Cost Entity Instances” on page 378. Another related discussion is “Fixed ID Keys Link a Cost Data Object to Another Object” on page 375.

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.

Note: See “Copying Custom Properties from Cost Data Objects to Cost Entity Instances” on page 378 for more details about `setFieldValue`.

In the policy auto line, the `setSpecificFieldsOnCost` method looks like:

```

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<com.guidewire.commons.entity.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
}

```

Note: For more discussion on this pattern and the `isCostVersionListForVehicle` method, See “Copying Custom Properties from Cost Data Objects to Cost Entity Instances” on page 378.

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}
}

```

Note: See “Key Values for Each Cost Data Subclass” on page 377.

Rate Slice Details for PersonalAutoCovCostData

The PASysTableRatingEngine class implements the `rateSlice` method with the following general algorithm. The structure is similar to other lines of business:

1. If the slice is canceled, do nothing.
2. 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.
3. 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.

It might be useful to see more details of rating of one of the coverages. In this case, let us look at the standard collision 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 rateVehicleCoverage_impl(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 based on things like:

- 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:

- Start date = the start of the slice to rate
- End date = the start of the next slice date

The implementation looks like:

```
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.FixedId)  
    populateCostData(cost, baseRate, adjRate)  
    return cost  
}
```

This method calls a standard `populateCostData` method that is common to both vehicle-level and line-level coverage costs. This 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, RoundingMode.HALF_UP)  
    cost.copyStandardColumnsToActualColumns()  
}
```

The end result of the `rateVehicleCoverage` call is a fully-populated `PersonalVehicleCovCostData`. The code can now add it 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 converts the cost data object to a cost entity instance and persists it to the database.

Rate Window Method for PersonalAutoCovCostData

The `rateWindow` method in `PASysTableRatingEngine` rates the following:

- 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. See the following topics:

- “Workers’ Compensation Rating” on page 398
- “Inland Marine Rating” on page 400
- “General Liability Rating” on page 400

Workers' Compensation Rating

Rating in workers' compensation is different from other lines of business. This is mostly because regulators tend to mandate the rating algorithm. Typically regulators require that companies rate the policies by rating period rather than by what PolicyCenter calls *slice dates* in most other lines of business. For more about the default algorithm used for other lines of business, see “The Rating Framework” on page 353.

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, if an anniversary date does not equal the period effective date. There are other possible reasons, such as re-rating the policy midterm to give the insured new rates mandated by the state. Each of these reasons to split the period causes:

- PolicyCenter creates a new `RatingPeriodSplitDate` object, sometimes called a *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 does the following things:

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
```

In this example:

- `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 class codes. 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.

An Algorithm Defined By Tables

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.

Note: 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:

- 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 `InlandMarlineLine` 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`.

Note: 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 people normally call *exposures*. General liability exposures must be viewed and edited in unsliced (windowed) mode, similar to class codes in workers' compensation line of business. However, other types of costs are rate-scalable costs with effective dates just like standard lines of business.

Note: For conceptual information on slice mode and window mode, see "Structure of Revisioning Across Effective Time" on page 497 in the *Application Guide*.

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).

Note: The structure of general liability data also ensures that the general liability rating engine cannot rate merely from a slice date forward. For related discussion, see "Can We Always Rate Only From the Current Slice Forward?" on page 362.

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 *Auditible* property has the value *false*, then this exposure is rate scalable. Note that the *Auditible* 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.

IMPORTANT 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`.

Reinsurance Integration

PolicyCenter supports reinsurance handling through integration of Guidewire Reinsurance Management or your own reinsurance management system.

IMPORTANT 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.

This topic includes:

- “Reinsurance Integration Overview” on page 403
- “Reinsurance Data Model” on page 405
- “Reinsurance Plugin” on page 407
- “Reinsurance Configuration Plugin” on page 413
- “Reinsurance Ceding Plugin” on page 416
- “Reinsurance Coverage Web Service” on page 418

See also

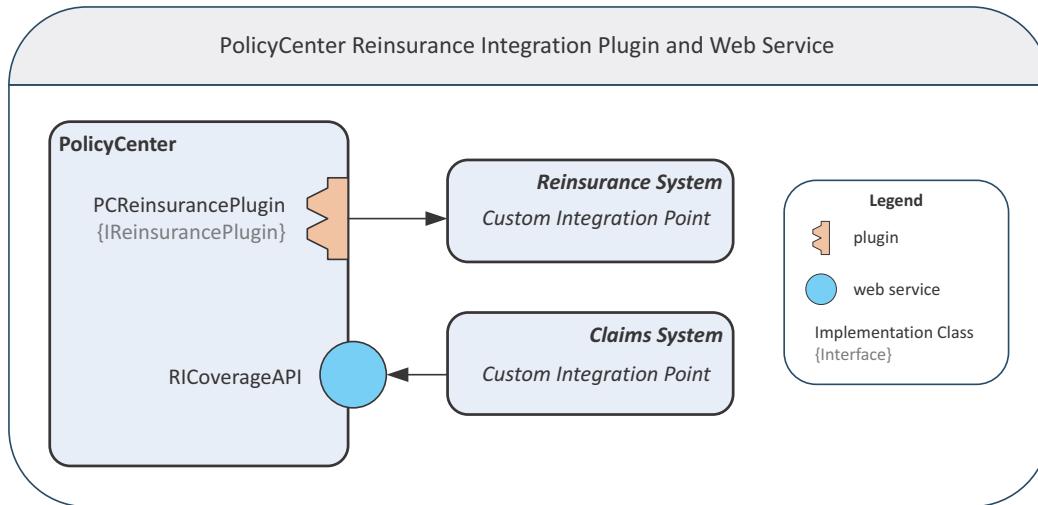
- “Reinsurance Management Concepts” on page 617 in the *Application Guide*
- “Configuring Multicurrency and Reinsurance” on page 603 in the *Configuration Guide*

Reinsurance Integration Overview

Reinsurance is insurance risk transferred to another insurance company for all or part of an assumed liability. Reinsurance can be thought of as 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.

Note: The insurance company directly selling the policy is also known in the industry as the carrier, the reinsured, or the ceding company. This topic uses the term *carrier* to refer to this company. 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.

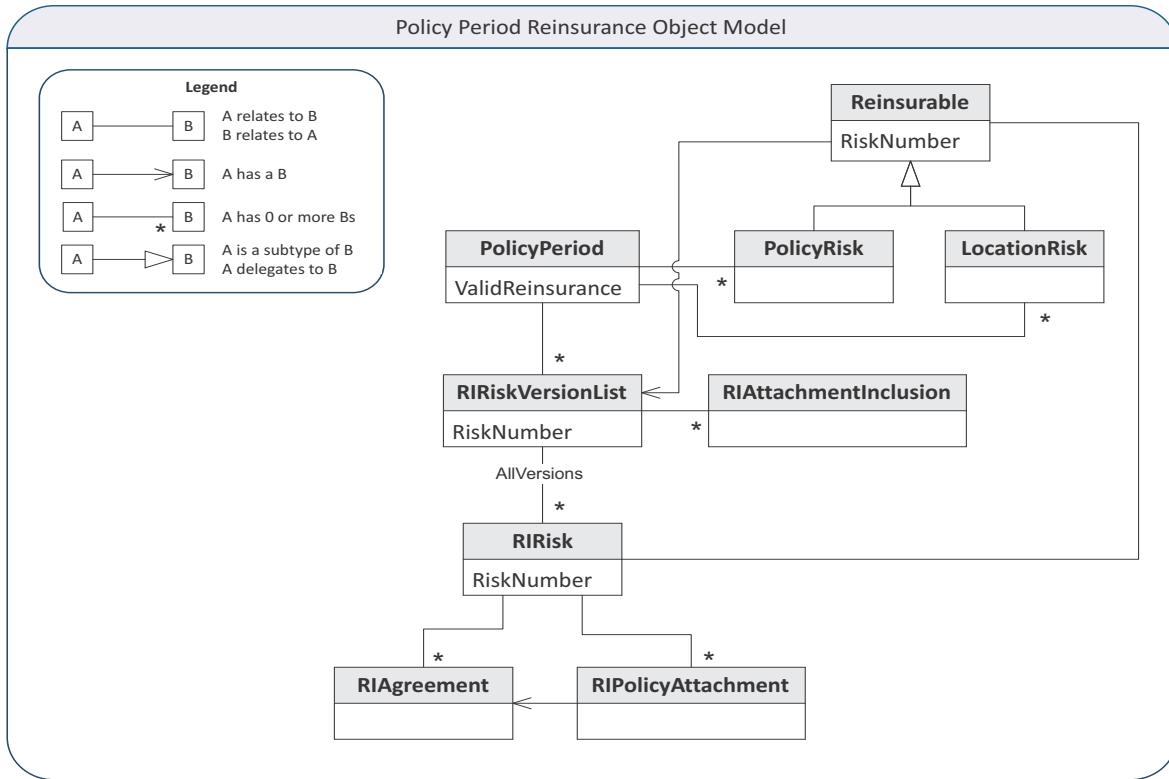


The plugin lets you integrate a reinsurance system with PolicyCenter. The web service lets you integrate PolicyCenter with a claims system for the purposes of reinsurance coverage.

- `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. For more information, see “Reinsurance Plugin” on page 407.
- `RICoverageAPI` – Web service that lets claims systems, such as Guidewire ClaimCenter, retrieve reinsurance coverage information from policies in PolicyCenter. For more information, see “Reinsurance Coverage Web Service” on page 418.

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.

See also

- “Structure of Revisioning Across Effective Time” on page 497 in the *Application Guide*
- “Reinsurance Management Object Model” on page 581 in the *Configuration Guide*
- The PolicyCenter *Data Dictionary*

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.

IMPORTANT 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:

- **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.

For the full list of properties on RIRisk instances, see:

- “Reinsurance Management Screens” on page 649 in the *Application Guide*
- “Reinsurance Management Object Model” on page 581 in the *Configuration Guide*
- The PolicyCenter *Data Dictionary*

Methods on Risk Instances

The RIRisk entity has a number of methods. The methods for the reinsurance plugins and web service are:

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 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 AllRisks. If a PolicyPeriod has any related RIRiskVersionList instances in its AllRisks 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:

Property	Description
RiskNumber	Unique identifier of a reinsurable risk.
AllVersions	Array of RIRisk instances that represent different versions of the same reinsurable risk.
AttachmentInclusions	PolicyCenter creates an RIAttachmentInclusion if the inclusion status of the attachment differs from the default, Included. Therefore, only attachments that are excluded or have special acceptance have an attachment inclusion row. When you exclude an attachment, PolicyCenter creates an RIAttachmentInclusion with the status set to Excluded. See also "Editing Ceding Parameters" on page 649 in the <i>Application Guide</i> .
PolicyPeriod	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:

Method	Description
addToAttachmentInclusions	Adds an RIRisk to the array of RIAttachmentInclusion instances on this RIRiskVersionList, or removes one. The array name is AttachmentInclusions. An RIAttachmentInclusion relates an RIRisk on a policy period to a reinsurance agreement.
removeFromAttachmentInclusions	
addVersion	Adds an RIRisk to the array of RIRisks on this RIRiskVersionList. The array name is AllVersions.
endDate	Splits the active RIRisk by doing all of the following: <ul style="list-style-type: none"> Sets ExpirationDate to today on the current RIRisk. Creates a new RIRisk instance with the same RiskNumber as the original. Sets the EffectiveDate on the new instance to today. Replaces all remaining fields except ExpirationDate from the original. Adds the new instance to the same RIRiskVersionList as the original.
getRIRisk	Gets the RIRisk that is active today.
getVersionAsOfDate	Gets the RIRisk that is active for a specified date.

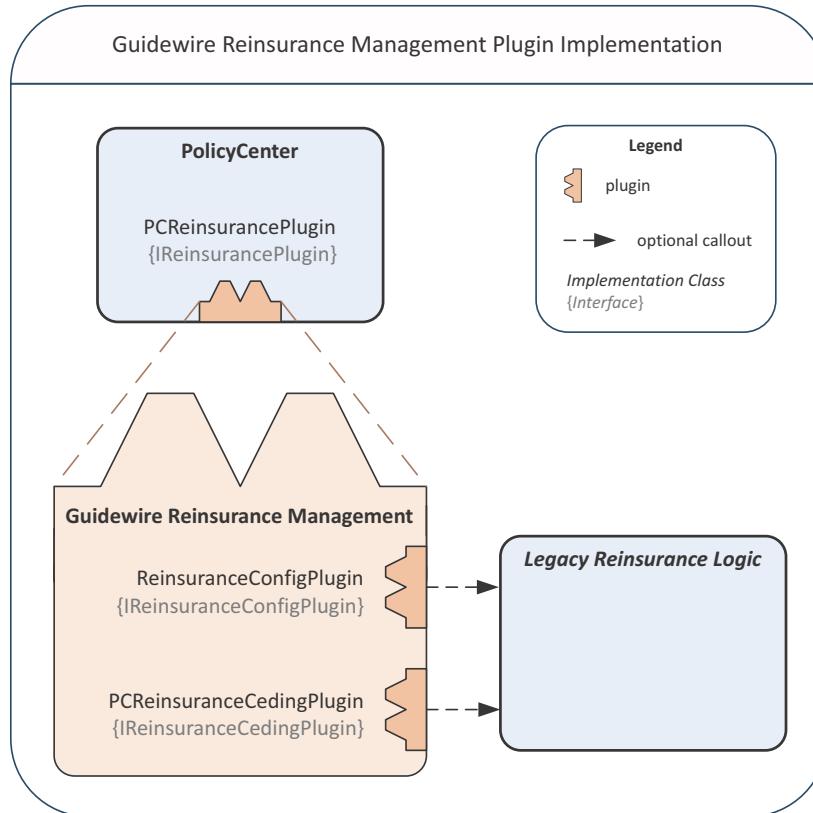
Reinsurance Plugin

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 and then opening IReinsurancePlugin.gwp.

In the default 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. This plugin integrates Guidewire Reinsurance Management with PolicyCenter. See “Reinsurance Plugin Implementation” on page 409.



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. See “Reinsurance Configuration Plugin” on page 413.
- `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. See “Reinsurance Ceding Plugin” on page 416.

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.

IMPORTANT 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 `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 Reinsurance Program Finder Interface

The `PCReinsurancePlugin.gs` plugin calls methods in implementation of the `RIProgramFinder` interface to select a reinsurance program for the current policy period. For example, the following code gets the `finder` object:

```
var finder = PCDependencies.getRIProgramFinder()
```

Programs for succeeding years may either be in draft status or not yet entered into PolicyCenter. The implementation of the `RIProgramFinder` interface 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 a carrier 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.

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` with the following methods:

- “Attaching Reinsurable Risks to Policy Periods” on page 410
- “Reattaching Risks” on page 411
- “Removing RIRisk Instances Associated with Reinsurable Risks” on page 411
- “Validating Risks” on page 411
- “Cleaning Up after Removing Reinsurables in a Renewal Job” on page 412
- “Binding Draft Reinsurance Version Lists Associated with a Branch” on page 412
- “Withdrawing Reinsurance Version Lists Associated with a Branch” on page 412
- “Whether Contacts Can Be Deleted” on page 412
- “Getting the Location Risk Group” on page 412
- “Setting the Location Risk Group” on page 412
- “Getting Risks in a Location Risk Group” on page 412

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 default 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

- “How PolicyCenter Attaches Agreements to Policies” on page 633 in the *Application Guide*
- “Adding Reinsurance to a Policy” on page 644 in the *Application Guide*
- “Choosing Reinsurance Programs for New Reinsurable Risks” on page 415

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

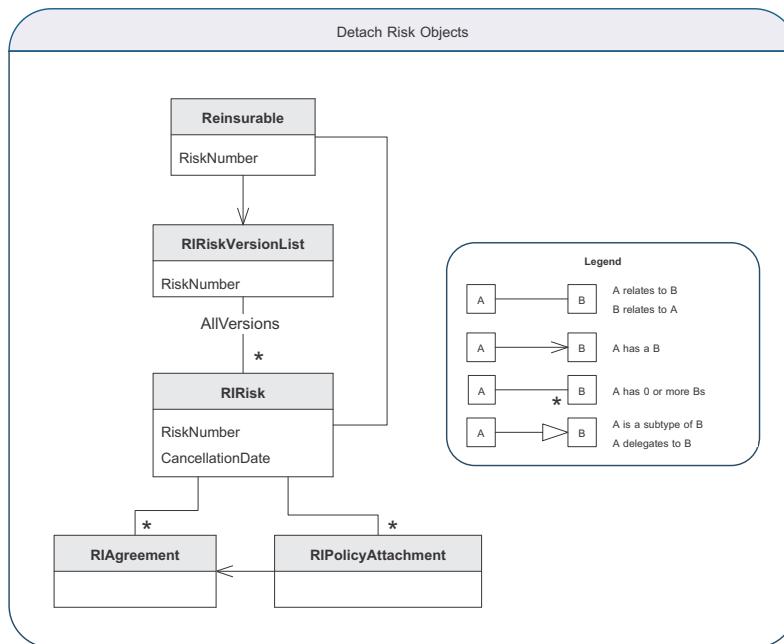
PolicyCenter calls the `detachRisk` method on `IReinsurancePlugin` to remove from the version list a risk that expires after the specified edit effective date of the policy period.

```
override function detachRisk(reinsurable : Reinsurable, branch : PolicyPeriod)
```

In the default implementation for a given `Reinsurable`, this method does the following to each `RIRisk` in the `RIRiskVersionList`:

1. Finds the one `RIRisk` that is effective at the edit effective date of the policy period.
2. Changes the `ExpirationDate` of that `RIRisk` to the edit effective date of the policy period.
3. 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

- “Performing Class-Based Validation” on page 71 in the *Rules 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

- “Customizing Behavior After Applying Changes from a Branch” on page 155

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 PolicyCenter Studio by navigating in the Project window to `configuration → config → Plugins → registry` and then opening `IReinsuranceConfigPlugin.gwp`.

See also

- “Configuring Multicurrency and Reinsurance” on page 603 in the *Configuration Guide*

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 carrier 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.

The `IReinsuranceConfigPlugin` has methods and properties for the following purposes:

- “Getting Default Gross Retention Amounts for Reinsurable Risks” on page 413
- “Getting Inclusion Types for Risks and Related Reinsurance Agreements” on page 414
- “Getting Override Ceded Amounts for Surplus Reinsurance Treaties” on page 414
- “Effective Time for New Reinsurance Programs and Agreements” on page 414
- “Generating Risk Numbers for New Reinsurable Risks” on page 414
- “Getting the Targeted Maximum Retention for a Reinsurable Risk” on page 415
- “Whether to Generate Reinsurable Risks on Policy Periods” on page 415
- “Whether a Program Covers a Reinsurable” on page 415
- “Choosing Reinsurance Programs for New Reinsurable Risks” on page 415

Getting Default Gross Retention Amounts for Reinsurable Risks

PolicyCenter calls the `getDefaultGrossRetention` 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
```

This 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
```

This 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
```

This 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 cede 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
```

This method returns a `Date` with the time portion specified. Use the value returned as the time portion of reinsurance effective dates and intervals.

The default 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
```

This method returns a `java.lang.String` with the unique number of a reinsurable risk.

The default 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 `ReinsuranceEffectiveTime` 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:

Parameter	Description
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 values are: <ul style="list-style-type: none">• ACTIVE_PROGRAMS• DRAFT_PROGRAMS• PRIOR_YEAR_PROGRAMS

This method returns an RIProgram 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 PolicyCenter Studio by navigating in the Project window to `configuration → config → Plugins → registry` and then opening `IReinsuranceCedingPlugin.gwp`.

IMPORTANT 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.

See also

- To learn more about the business process to cede premiums, see “How PolicyCenter Attaches Agreements to Policies” on page 633 in the *Application Guide*.

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. For an example of how PolicyCenter calculates the ceded premium, see “Calculating Ceded Premiums” on page 637 in the *Application Guide*.

This implementation of the plugin does not calculate ceded premiums for non-proportional treaties. However, you can add this calculation to this plugin.

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.

The `IReinsuranceCedingPlugin` has methods for the following purposes:

- “Enqueuing Policy Periods for Ceding Calculations” on page 417
- “Calculating Ceded Premiums for Policy Periods” on page 417
- “Whether to Recalculate Ceding Amounts” on page 418
- “Which User is Responsible for Reinsurance Program Changes” on page 418
- “Logging Errors for Invalid Reinsurance Programs” on page 418

See also

- “Batch Processing” on page 111 in the *System Administration Guide*

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:

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> type list
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 : RIRecalcReason, comment : String, updateUser : User)
```

The parameters are:

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>RIRecalcReason</code> type list.

Parameter	Description
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 RIRisk.

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
```

This method takes a single parameter, an `RICedingWorkItem`, which is the item to evaluate for recalculation.

This method returns a `boolean`. 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:

Parameter	Description
<code>period</code>	The period that the program affected
<code>dirtyPrograms</code>	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>)
```

This 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. This 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` → `Classes` → `webservice` → `pc` → `pc800` → `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 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

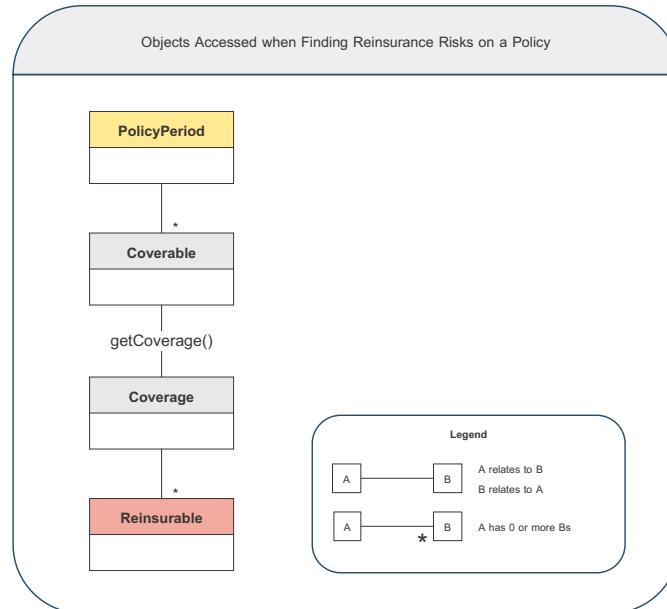
The **findRIPolicyRisk** method finds reinsurable risks of the specified reinsurance coverage group type that are attached to the policy on a particular date. A reinsurable risk exists on a policy. For example, the property coverages at a particular location constitute a reinsurable risk.

```
function findRIPolicyRisk(policyNumber : String, coverageCode : String, date : Date) : RIRiskInfo
```

The parameters are:

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 a **RIRiskInfo** instance. For more information, see “Reinsurance Risk Information” on page 420.

Finding Reinsurable Risks by Coverable on Policy

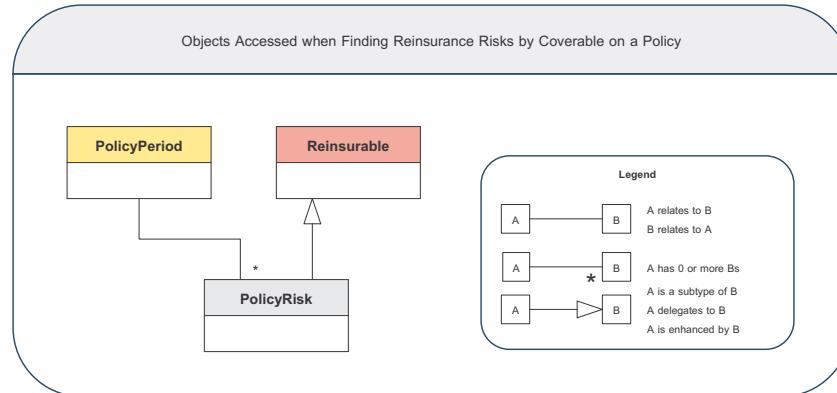
The **findRIPolicyRiskByCoverableID** method finds reinsurable risks of the specified reinsurance coverage group type that are attached to a specified coverable on the policy on a particular date.

```
function findRIRiskByCoverableID(policyNumber : String, coverableID : String, coverageCode : String, date : Date) : RIRiskInfo
```

The parameters are:

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 a `RIRiskInfo` instance. For more information, see “Reinsurance Risk Information” on page 420.

Reinsurance Risk Information

Both the `findRIPolicyRisk` and `findRIPolicyRiskByCoverableID` methods return risk information in an `RIRiskInfo` object. In the base configuration, these methods obtain the `RIRiskInfo` by calling the `findReinsuranceRiskInfo` method on the `IReinsurancePlugin` plugin.

The `RIRiskInfo` object has the following fields:

Field	Description
Description	A description such as <i>Property coverage for Location 1</i> or <i>Auto liability coverage for Personal Auto line</i> . The pattern for the description is: <code>CoveragePattern.Name coverage for Reinsurable.DisplayName</code>
RIRiskID	The PublicID of the <code>RIRisk</code> attached to the <code>Reinsurable</code> .
Agreements	An array of reinsurance agreements, as described below.

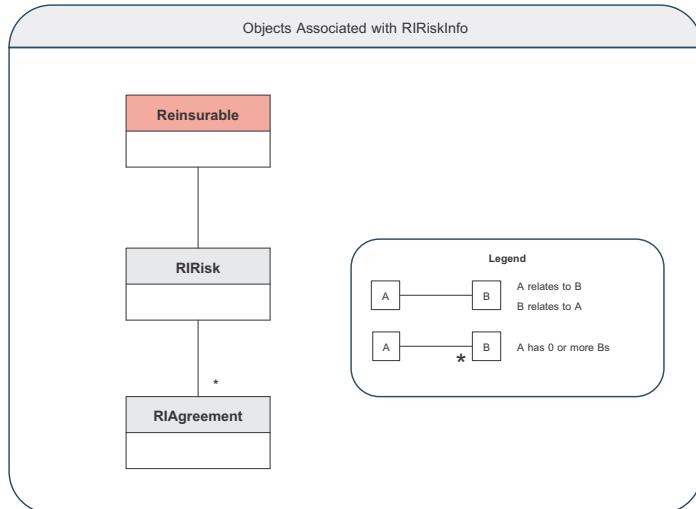
The `RIRiskInfo` object contains information about the reinsurance agreements. The object contains no agreement that is specifically excluded nor any proportional agreement with a 0% share. The object includes all non-proportional agreements. This object does not contain agreements in draft status.

The `RIRiskInfo` object contains the following information for each `RIAgreement`:

Field	Description
AgreementNumber	The <code>AgreementNumber</code> of the agreement.
Name	The <code>Name</code> of the agreement.
Type	The <code>Subtype</code> of the agreement, such as <code>quota share</code> or <code>surplus</code> .

Field	Description
CededShare	For non-proportional agreements only, this is the CededShare percentage of the attachment.
Comments	The Comments of the agreement.
ProportionalPercentage	For proportional agreements, this is the proportional percentage of the risk that the attachment takes. In PolicyCenter, this is the Prop % column for an agreement.
AttachmentPoint	For non-proportional agreements only, the AttachmentPoint of the agreement.
AttachmentPointIndexed	For non-proportional agreements only, whether the attachment point is LimitIndexed.
TopOfLayer	For non-proportional agreements only, the CoverageLimit.
TopOfLayerIndexed	For non-proportional agreements only, the attachmendindexed field on an agreement.
RecoveryLimit	For proportional agreements, the CededRisk of the attachment. For non-proportional agreements, the amount of reinsurance specified by the agreement. This is a calculated percentage.
NotificationThreshold	For per risk agreements only, the NotificationThreshold.
EffectiveDate	The EffectiveDate.
ExpirationDate	The ExpirationDate on the agreement.
Draft	For treaties, this is true if the reinsurance program that contains the agreement is not yet active. For facultative agreements, this is true if the facultative agreement is not yet active.

The following illustration shows some of the objects associated with the RIRiskInfo object.



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.

This topic includes:

- “Forms Integration Overview” on page 423
- “Forms Inferences Classes” on page 424
- “Forms Messaging” on page 429

See also

- “Policy Forms” on page 471 in the *Application Guide*
- “Messaging and Events” on page 289

Forms Integration Overview

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.

Forms Inferences Classes

You configure basic forms settings in Studio 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 the following:

```
FormInferenceEngine.Instance.inferPreBindForms(_branch)
    _branch.addEvent("IssueSubmission")
```

See also

- “Creating Inference Data and XML” on page 424
- “Forms Messaging” on page 429

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.

See also

- For more information about using `XmlNode` objects, see “Gosu and XML” on page 271 in the *Gosu Reference Guide*.

Inference Data

The important method for creating inference data is `populateInferenceData`, called by the forms inference engine to populate this instance with the appropriate data from the policy graph. This method is called immediately after the instance of the `FormData` is created.

One parameter to the method is an *inference context object* (`FormInferenceContext`). An inference context object contains important information, such as:

- The `PolicyPeriod`, in the `Period` property of the context object
- The set of forms in the group, in the `Patterns` property of the context object

The other parameter is a set of available states that contains all states in which this form was found to be available. If a state-specific form is available that replaces the national version of a form, for a national form those states automatically do not appear in the set. This auto-filtering feature by jurisdiction based on group code settings in the product model is called *jurisdictional replacement*.

The recommend approach for implementing this is to generate and then store inference data in a private variable so it can be read by other methods in your class. For example, a built-in personal auto form can create towing labor coverage data like the following in its `populateInferenceData` method:

```
var towingInfoSet = mapVehicles(context, \ v -> v.PATowingLaborCovExists, \ v -> createTowingInfo(v))

_towingInfo = towingInfoSet.toList().sortBy(\ info -> info.Vin)
```

In the preceding 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 list (`java.util.List`) and sorted by the vehicle identification number.

Note: The final result is a list (not an `XMLNode` nor a XML-formatted text). You must store it in a private variable that you define in your own `FormData` subclass.

See also

- For more information about `mapArrayToSet`, see “Form Data Helper Functions” on page 427.

XML of Inference Data

The important method for exporting the inference data is the `addDataToContentNodeForExport` method. Your method must take your inference data stored in private variables and add child XML nodes to the `XMLNode` that is provided as a parameter.

For example, the same built-in personal auto form described earlier uses the following code to generate XML data from that list in the `_towingInfo` variable:

```
override function addDataToContentNodeForExport(contentNode: XMLNode) : void
{
    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. See “Determining Whether to Add or Reprint a Form” on page 426.

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. See “Handling Multiple Instances of One Form” on page 428 for important information on this topic.

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 "true".
<code>ignoreAttributes</code>	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.
<code>ignoreAllAttributes</code>	Indicates to ignore all attributes on a node during comparison. Set this attribute value to "true" for this behavior.
<code>ignoreText</code>	Indicates to ignore the text of a node during comparison. Set this attribute value to "true" for this behavior.
<code>ignoreChildren</code>	Indicates to ignore the children of a node during comparison. Set this attribute value to "true" for this behavior.

The most important attributes and methods are `ignoreAll` and `ignoreAttributes`. The other methods exist primarily for completeness.

See also

- For more information about using `XmlNode` objects, see “Gosu and XML” on page 271 in the *Gosu Reference Guide*.

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 Studio, 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.

For example, for the example inference class earlier in this section, you could use code such as:

```
override property get ShouldBeAdded() : boolean
{
    return !_towingInfo.Empty
}
```

In other words, this 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>createTextNode</code>	Creates an XML node with the specified name and text content
<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>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>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>addDataToContentNodeForExport</code> method to output your data.
<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.
<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>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. See "Handling Multiple Instances of One Form" on page 428.

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.

See also

- “Gosu Generics” on page 243 in the *Gosu Reference Guide*

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.

To support form multiplicity

1. In the data model, create a subtype of `FormAssociation`. To continue the earlier example, perhaps a new entity called `PAVehicleFormAssociation`.
2. On that entity, add property with a foreign key link to the desired type, such as a personal vehicle. For this example, 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 (see “Gosu Generics” on page 243 in the *Gosu Reference Guide*) to abstract this class for your linked-to type. In this example, use the following syntax to define the inference class:

```
class MyPAVehicleForm extends AbstractMultipleCopiesForm<PersonalVehicle>
```
4. In your inference class, do not implement the `populateInferenceData` method. That method is only used if you do not use form multiplicity.
5. Instead of the `populateInferenceData` method, add to your inference class the `getEntities` method, which must return a list containing one or more entities of your desired type. For this example, it would be a list of all personal vehicles on the policy. 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.
6. Add to your inference class the `FormAssociationPropertyName` method, which simply returns the name of the property in your `FormAssociation` subtype, as a `String`. For the earlier example, pass the `String` value “`Vehicle`” because the property name is `Vehicle`.
7. In your inference class XML generation method `addDataToContentNodeForExport`, export the data for only one entity in your generated list of entities. Your code refers to the current one using the special private variable called `_entity`, which is set up automatically for you by the `AbstractMultipleCopiesForm`. In this example, the `_entity` variable contains one `PersonalVehicle` from the list returned in your `getEntities` method. Your `addDataToContentNodeForExport` would add child XML nodes for that vehicle only.
8. In your inference class create the `createFormAssociation` method to instantiate one version of your form association subtype. It can use simple code such as:

```
override protected function createFormAssociation( form : Form ) : FormAssociation {  
    return new PAVehicleFormAssociation(form.Branch)  
}
```

Remember that in your messaging code, you must access `Form.FormAssociations` to get form associations that link to the associated entity. See “Forms Messaging” on page 429 for related information.

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. See “List of Messaging Events in PolicyCenter” on page 309 for a full list.

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. See “Creating Inference Data and XML” on page 424 for details of creating that data. You can get the cached XML data using code like:

```
cachedXML = myForm.FormTextData.TextData
```

You might make your message payload simply the XML payload generated from that `XmlNode` such as the 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. For multiplicity, see “Handling Multiple Instances of One Form” on page 428.

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.

See also

- For more information on XMLnode objects, see “Gosu and XML” on page 271 in the *Gosu Reference Guide*.

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 191

Policy Difference Customization

Sometimes PolicyCenter compares two policy *branches*. 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 customize the appearance of policy differences and the underlying mechanism that calculates the differences between two policy branches.

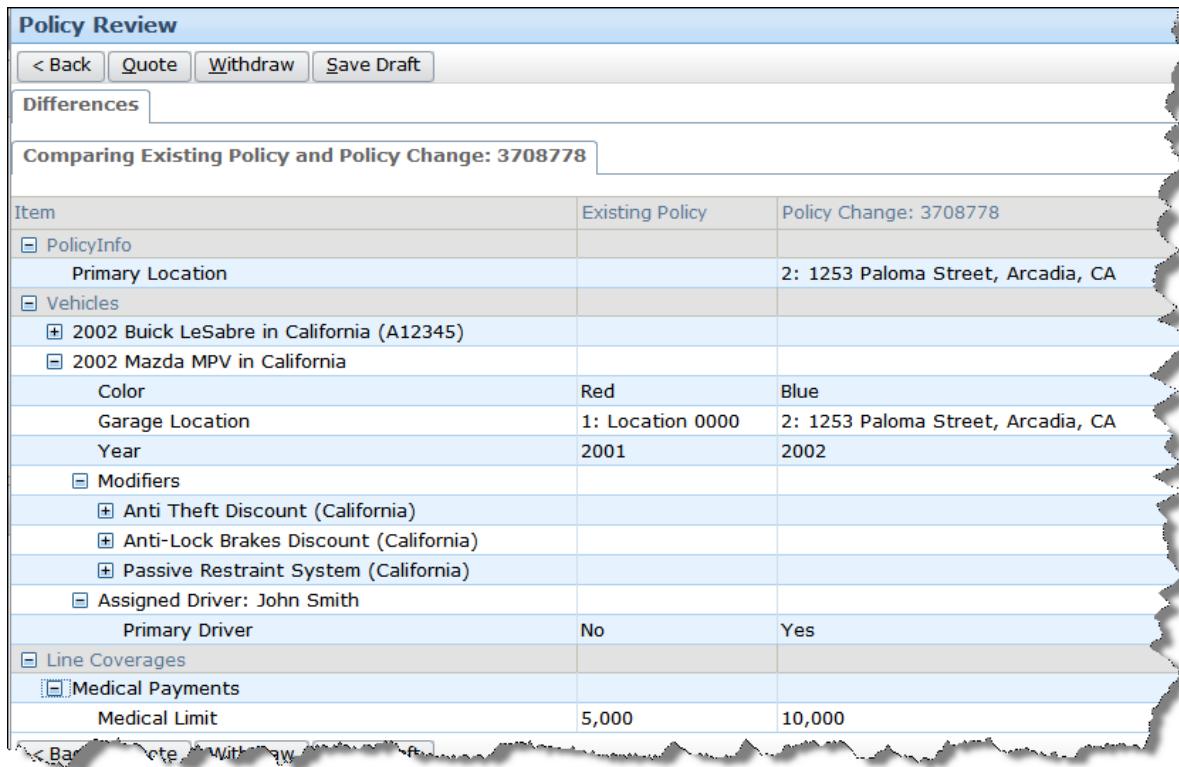
This topic includes:

- “Policy Difference Overview” on page 431
- “Difference Tree XML Configuration” on page 441
- “Customizing Differences for New Lines of Business” on page 449
- “Customizing Personal Auto Line of Business” on page 450
- “APIs for Calculating Differences” on page 451

Policy Difference Overview

There are several places in the PolicyCenter user interface where you can view comparisons between policy periods. You can customize the appearance of these differences and customize how PolicyCenter calculates differences between policy periods.

The most common user interface for policy differences is the policy review screen that you can view as part of a policy change job:



The screenshot shows the 'Policy Review' interface. At the top, there are buttons for '< Back', 'Quote', 'Withdraw', and 'Save Draft'. Below that is a section titled 'Differences' with the sub-section 'Comparing Existing Policy and Policy Change: 3708778'. The main table compares 'Existing Policy' and 'Policy Change: 3708778' across various items. Some items have expandable sections:

Item	Existing Policy	Policy Change: 3708778
PolicyInfo		
Primary Location		2: 1253 Paloma Street, Arcadia, CA
Vehicles		
2002 Buick LeSabre in California (A12345)		
2002 Mazda MPV in California		
Color	Red	Blue
Garage Location	1: Location 0000	2: 1253 Paloma Street, Arcadia, CA
Year	2001	2002
Modifiers		
Anti Theft Discount (California)		
Anti-Lock Brakes Discount (California)		
Passive Restraint System (California)		
Assigned Driver: John Smith		
Primary Driver	No	Yes
Line Coverages		
Medical Payments		
Medical Limit	5,000	10,000

From a configuration perspective, the top-level trigger for most of this code is PolicyCenter application logic calling the registered plugin for the `IPolicyPeriodDiffPlugin` interface. This plugin has two methods, called `filterDiffItems` and `compareBranches`, and the user interface calls one method or the other in different places in the application flow. To write Gosu code that triggers the plugin to generate and filter differences, see “APIs for Calculating Differences” on page 451.

Both plugin methods help generate and filter a list of differences, each of which is encapsulated in a difference item (`DiffItem`) entity. The difference item list represents the similarities and differences between the items that determine what entities added or removed and what properties changed.

PolicyCenter chooses which method to call based on the context of the request. There are two different strategies that PolicyCenter uses to generate and filter difference items:

Difference Item Generation Strategy	Description	What generates differences?	What filters differences?
Database generation of difference items	If PolicyCenter compares a policy period to the revision that it was based on, it can calculate changed properties and entities directly from the database. This process is very efficient and accurate. The fundamental task of your code is to filter out irrelevant differences. This is the PolicyCenter approach for policy changes and some other application flows.	PolicyCenter, generally speaking. Think of the primary difference generation for this strategy as a low level database-driven query. However, there is a minor rare exception. For Workers' Comp and General Liability, the low level database code does not generate exposure splits. To handle that, the <code>filterDiffItems</code> method in the plugin generates additional differences. See the <code>WCDiffHelper</code> and <code>GLDiffHelper</code> classes. That code removes all exposures that split, and then adds them back again.	The original entrypoint is the <code>IPolicyPeriodDiffPlugin</code> plugin method <code>filterDiffItems</code> . Only filter out differences in that method if you are sure no PolicyCenter code ever needs those differences. Multiple points in the application might need the list of differences: <ul style="list-style-type: none"> Logic that merges or applies changes from one revision to another. User interface code to display differences. You can filter dynamically based on the difference reason, which is a method parameter with the type <code>DiffReason</code> .
Compare two branch graphs	For multi-version jobs and some other contexts, Gosu code that the policy period difference plugin triggers compares two arbitrary branch graphs and generates a list of differences. The list of differences represents the similarities and differences between the items. In other words, Gosu code determine what was added, removed, or changed.	The policy period difference plugin in the <code>compareBranches</code> method	This strategy is mostly about generating difference items not filtering them. However, this approach uses a secondary filtering process. For more details about this filtering, see "Difference Helper Classes" on page 435 and "Difference Utility Classes" on page 436.

The following table lists various places PolicyCenter checks for differences between policy periods, how it is implemented, and which of the earlier approaches are used.

PolicyCenter uses policy comparisons in the following places in the product:

Reason to check for differences	Strategy for generating differences	Description
Policy change transaction	Database generation	The Policy Review page lets you see what changed between the active branch and the branch upon which it was based.
Renewal transaction	Database generation	The Policy Review page lets you see what changed between the active branch and the branch upon which it was based.
Rewrite transaction	Database generation	The Policy Review page lets you see what changed between the active branch and the branch upon which it was based.

Reason to check for differences	Strategy for generating differences	Description
Out-of-sequence jobs (of multiple types)	Database generation	PolicyCenter must check whether a job is out of sequence (see “Out-of-sequence Jobs” on page 511 in the <i>Application Guide</i>) 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	PolicyCenter uses differences in two places in the preemption flow: <ul style="list-style-type: none"> First, PolicyCenter must check whether a job was preempted (see “Preempted Jobs” on page 513 in the <i>Application Guide</i>). If so, PolicyCenter must display and apply 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, there can be changes that 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. For the typical case, you compare 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 none are based directly on each other. Some data is simplified or omitted for display. For example, for personal auto line, the interface compares vehicles, coverage amounts, and resulting costs in the quote. However, 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 used in the policy review and out-of-sequence cases. Instead, PolicyCenter uses the same system used in the multi-version difference generation system.

Customizing How to Display Difference Items to Users

The tables in the previous topic describe the primary generation and filtering of policy differences. However, 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. Or, 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.

See also

- To learn how to configure what differences users can view, see “Difference Tree XML Configuration” on page 441.

Difference Item Subclasses

There are several types of difference items, all of which are subclasses of the `DiffItem` class, described in the following table. Some difference items classes are only created in the reference implementation in some types of comparisons, although you could customize it to use any if desired. The rightmost columns in the following table indicate which ones are created in the reference implementation in different contexts.

Class	Description	Created if comparing two branch graphs (for example, multi-version)	Created by database generation of difference items (for example, policy change)
<code>DiffAdd</code>	An entity was added that did not exist before	Yes	Yes
<code>DiffRemove</code>	An entity was removed that existed before	Yes	Yes
<code>DiffProperty</code>	A property changed in an entity.	Yes	Yes
<code>DiffWindow</code>	<p>A change in window mode to an entity'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 would not be used for a typical change, which would normally done as a slice change (see the row for <code>DiffSlice</code>). For more about these topics, see "Slice Mode and Window Mode Overview" on page 499 in the <i>Application Guide</i>.</p> <p>This would not appear in a multiple revision job (such as a multiple revision quote).</p> <p>However, 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.</p>	Yes, but not in multiple revision jobs (see notes)	Yes

You can create these difference items directly using their constructors, with code such as:

```
DiffAdd diff = new DiffAdd(newElem);
diff.setPath(ctx.getPath());
```

Alternatively, use APIs to create difference items by comparing entities in two branches. See the following subtopics:

- “Difference Helper Classes” on page 435
- “Difference Utility Classes” on page 436
- “Customizing Personal Auto Line of Business” on page 450.

Difference Helper Classes

One of the two difference item strategies is to compare two arbitrary branches using the policy period difference plugin `compareBranches` method.

In the built-in implementation of this plugin, the default logic of `compareBranches` method compares policies using helper classes that look at details for each line of business. Each built-in line of business has one of these difference helper classes, named with the line of business in the class name.

For example, the reference implementation of the plugin uses each line of business and calls its `createPolicyLineDiffHelper`

method. It is responsible for returning the correct difference helper class. 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( l1 -> l1.Subtype.Code == line1.Subtype.Code )
    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>
        }
    }
}

```

For more information about the methods provided in the built-in personal auto `PADiffHelper` class, see “Customizing Personal Auto Line of Business” on page 450.

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 (subclass) the built-in `DiffHelper` class. Your subclass must encapsulate the line-specific logic and call other classes like `DiffUtils` as necessary. See the following section for more about `DiffUtils`.

See also

To learn how to configure what differences users can view, see “Difference Tree XML Configuration” on page 441.

Difference Utility Classes

PolicyCenter includes a Gosu class called `DiffUtil` that is used by other built-in classes to compare two branches and to help with filtering a list of different items.

Note: This topic describes filtering difference items during the process of creating difference items. To configure what differences display in the tree view and how they appear, customize a difference tree configuration XML file. For more info, see “Difference Tree XML Configuration” on page 441.

One of its important tasks is to compare two branch graphs and generate difference items. To do this task as part of creating an instance of `DiffUtil`, you specify the following information to an instance of the `DiffUtil` class:

- **A bean matcher instance** – A bean matcher is an object that compares two objects and determine if they are equal by looking at property data rather than at a `fixedID` or primary key. For example, you can tell two cars are the same by checking their vehicle identification number (VIN). You can tell two contacts (`PolicyContact` objects) are the same by checking the name and address. You do not have to write one of these on your own, as discussed further in the following discussion, although you can customize the existing logic. For details, see “Compare Individual Objects with Matchers” on page 437. The built-in PolicyCenter bean matcher handles objects in the PolicyCenter reference implementation. You probably do not need a totally new bean matcher subclass. For related information, see “Compare Individual Objects with Matchers” on page 437.
- **Set of excluded types** – Each `DiffUtil` instance includes a list of types to exclude. In the built-in implementation, PolicyCenter uses `Transaction`, `PACost`, `WCCost`, `BOPCost`.
- **Set of excluded properties** – Each `DiffUtil` instance includes a list of properties to exclude.
- **Set of included properties** – Each `DiffUtil` instance adds a list of properties to include, even if excluded by other rules. In the built-in implementation, PolicyCenter includes the cost property `Amount`.

The most important methods on the `DiffUtils` class include:

- `compareBeans` – Compare two entity graphs. This compares all properties and traverses all arrays and links up to a specified depth, which specifies how many jumps down in the hierarchy to examine.
- `compareField` – Compare two entity graphs starting at a specific property on the entities. The property can be a column, link, or array. From that property, compares all properties and traverses all arrays and links up to a specified depth, which specifies how many jumps down in the hierarchy to examine. Pass the property as a `IEntityPropertyInfo` object, which you can get from the type system using syntax such as:

```
PersonalAutoLine.TypeInfo.getProperty( "PALineCoverages" )
```

To customize the built-in behavior, you have two options. You can either modify `DiffUtils` directly, which might make sense if you want to always exclude or include properties for all lines of business. Or, you can add additional logic to the class that instantiates `DiffUtils`. For example, for personal auto (PA) line of business, you can modify the `PADiffHelper` class.

Difference Utilities Example

The following example code instantiates the `DiffUtils` class and compares two business lines:

```
// create a new instance of DiffUtils
var diffUtils = new DiffUtils()

// include the cost properties
diffUtils.includeFields(line1)

//Calculate (add) line coverage diffs
var paDiffs = diffUtils.compareField( line1, line2,
    PersonalAutoLine.TypeInfo.getProperty( "PALineCovverages"), 2)

//Calculate (add) vehicle diffs
paDiffs.addAll(diffUtils.compareField( line1, line2,
    PersonalAutoLine.TypeInfo.getProperty( "Vehicles"), 2))
```

Compare Individual Objects with Matchers

There are a few parts of PolicyCenter that need specialized logic to answer the question, “do these two objects represent the exact same thing?”. These include:

- The policy difference page. For example, in a policy change job, comparing a policy to the original. Other situations use this same interface, 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 thing, PolicyCenter runs code that answers the question definitively. For example, to compare two cars, you might compare the unique Vehicle Identification Numbers (VIN numbers). If the VIN numbers match, they represent the same car, independent of the number of other properties on the car that might be different.

There are two types of matching APIs in PolicyCenter:

- Delegate-based matching classes
- PCBeanMatcher, which is an older style of matching class

How the Delegate-based Matchers Work

The delegate-based matchers use a delegate interface to define the required methods. The delegate interface depends on what type of object it is:

- For effective-dated subobjects in a `PolicyPeriod` graph, the matchers use the `EffDatedLogicalMatcher` interface.
- For all other entity types, the matchers use the `LogicalMatcher` interface.

To define a new matcher, use the right interface and define the necessary methods. You must be able to know the set of fields that uniquely identify an entity. For example, the `LogicalMatcher` interface has a method called `genKey` that generates a unique key that can be compared to determine if the objects match.

If the keys for two objects do not match, they are definitely not the same object.

If the keys for two objects match, they might or might not be the same object. To determine for sure, PolicyCenter calls the `isLogicalMatchUntyped(KeyableBean bean)` method, which may do additional checks.

This method might check some set of parent foreign keys that must also match. For example, a `VehicleDriver` would match if the foreign keys to `PersonalVehicle` and `PolicyDriver` also match.

About 100 entity types now implement one of the matcher delegates and did not have explicit code in PCBeanMatcher. In contrast, the PCBeanMatcher matches about 30 entity types, most notably including Cost objects.

The base comparison of comparing Id and FixedID properties for all entities remains in the PCBeanMatcher class.

The default implementation of the delegate-based matchers relies on defining columns on the entity, including parent foreign key links to determine whether two objects match.

General tips with writing new matchers:

- You can match on whatever you wish, but be warned that not using true database-backed columns generally speaking is dangerous.

WARNING Avoid matching on non-database-backed columns, such as dynamically-generated Gosu properties. It is dangerous. Contact Guidewire Customer Support for guidance before implementing matchers like that.

- Column values must be immutable, values that never change over the life of the object. For example, an automobile VIN (Vehicle Identification Number) does not change over time. Editing a VIN never makes sense, other than to correct a typo.

Customize Which Classes Perform Matching

The policy difference system always initially uses the PCBeanMatcher class. Similarly, the PolicyCenter out-of-sequence handling system initially calls PCBeanMatcher.

However, the PCBeanMatcher always uses the LogicalMatcher delegate if one is available. The EffDatedLogicalMatcher class is a subtype of LogicalMatcher.

You can edit the PCBeanMatcher file, so you can change the matching behavior that you want to use when generating a list of differences.

Delegate Matchers and Copiers

Related to the delegate matcher classes are copier classes, which know how to completely duplicate an entity. 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.

However, in some cases, PolicyCenter also uses a copier. For example, an out-of-sequence job or a preemption job might need to do the following process, which is sometimes called *DuplicateAdd*:

1. Generate a policy difference between to branches to find added entity instances
2. Use matchers to see if new entity instances have a match somewhere in the graph
3. If they do match, use copiers to ensure that they are fully in sync as defined by each entity type.

Because of this, if you write a new matcher class, then you must write a copier class for your data. Otherwise, DuplicateAdd does not work as expected. PolicyCenter might extend an existing entity, but lose (not copy over) the new values.

IMPORTANT If you write a new matching class, you must write a new copier class.

See also

- “Configuring Copy Data in a Line of Business” on page 197 in the *Product Model Guide*

Bean Matchers

As mentioned in “Policy Difference Overview” on page 431, one of the two difference item strategies is to compare two arbitrary branches using the policy period difference plugin `compareBranches` method. During this process, the built-in implementation of the policy period difference plugin uses built-in classes called *bean matchers*, implemented in the class `PCBeanMatcher.gs`. This class compares two objects and determines if they are equal by looking at property data rather than by the `fixedID` property. This assists PolicyCenter in some edge cases to confirm whether two objects are the same.

Note: For more information about the fixed ID, see “What Is a Policy Revision?” on page 489 in the *Application Guide*.

If two entities have the same fixed ID (matching `fixedID` property values), then the entities represent the same item. If they do not match, then PolicyCenter generally can tell two cars are the same by comparing their vehicle identification numbers (VINs). Even if two entities match based on `fixedID` values, PolicyCenter might perform additional tests. For example, PolicyCenter determine two contacts (`PolicyContact` objects) are the same by comparing their name and address properties.

These comparisons are performed by the bean matcher. Typically, you do not have to write one of these on your own because PolicyCenter includes a subclass called `PCBeanMatcher` that handles common cases. However, you can customize the logic as necessary.

Note: Also see the discussion about delegate-based matchers in “Compare Individual Objects with Matchers” on page 437

Edit the `PCBeanMatcher.gs` file to customize this logic. The following is a simple example that checks whether two commercial properties are actually the same by checking the account locations:

```
private function hasCommercialPropertyMatch(b1 : KeyableBean, b2 : KeyableBean) : boolean {  
    switch(true){  
        case b1 typeis CPLocation:  
            return (b1 as CPLocation).PolicyLocation.AccountLocation ==  
                   (b2 as CPLocation).PolicyLocation.AccountLocation  
        case b1 typeis PolicyLocation:  
            return (b1 as PolicyLocation).AccountLocation == (b2 as PolicyLocation).AccountLocation  
        default:  
            return false  
    }  
}
```

Edit the existing methods or add more as appropriate.

If you add new methods to compare new types of objects, be sure to edit the top-level method in the class, called `doBeansMatch`. Test for your object’s type before calling your new method.

Generally speaking, PolicyCenter calls the bean matcher for two entities only if the fixed IDs do not match and the application needs to compare entities during a “compare branches” action.

However, some lines of business might use the bean matcher object even if generating differences from a based-on revision, as is done in policy change policy reviews. Some lines of business might need to compare certain non-revisioned objects or other objects that might be removed and then readded later. The built-in policy difference plugin calls the bean matcher (indirectly) for the personal auto line of business to compare policy contacts to see if they match. Any time code directly or indirectly calls the `compareField` or `compareBean` method on a `DiffUtils` object, PolicyCenter uses the bean matcher.

For more information about the two basic approaches in calculating differences, refer to the first table in the section “Policy Difference Overview” on page 431.

Important Files for Customizing Differences

The most important files for you to customize policy change differences are listed in the following table:

File name	Example	Which products and LOBs 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. For details, see “Difference Helper Classes” on page 435.
<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. For information, see “Compare Individual Objects with Matchers” on page 437 and “Customizing Differences for New Lines of Business” on page 449.
<i>file names vary</i>	n/a	All	Delegate-based matcher files for each type. See “Compare Individual Objects with Matchers” on page 437.
<code>difftree.xsd</code>	n/a	All	The XSD for node generation configuration XML files. For details, see “Difference Tree XML Configuration” on page 441.
<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. For details, see “Difference Tree XML Configuration” on page 441.
<code>DiffTreePanelSet.pcf</code>	n/a	All except BOP and WC. They use a legacy approach to differences display. (see later rows in this table)	Displays the differences tree based on settings in the XML file. For details, see “Difference Tree XML Configuration” on page 441.
<code>DifferencesPanelSet.pcf</code>	n/a	All. However, BOP and WC call out to <code>ComparisonCV.pcf</code> and <code>DifferenceCV.pcf</code> and use a legacy approach to differences display.	Displays the differences tree based on settings in the XML file. For details, see “Difference Tree XML Configuration” on page 441.

File name	Example	Which products and LOBs use the file?	Description
ComparisonCV.pcf	n/a	BOP and WC for legacy display. If the XML file is not configured for a product, PolicyCenter uses this.	For multi-version policy comparisons and similar application logic, this PCF page displays the differences.
DifferenceCV.pcf	n/a	BOP and WC for legacy display. If the XML file is not configured for a product, PolicyCenter uses this.	For policy change comparisons and similar application logic, this PCF file displays DiffItem objects in separate list views by the status of the entity: added, removed, or changed. Additional tabs may additionally appear and show non-slice changes and window mode changes.

Filtering Difference Items (After Database Generation)

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.Bean typeis TerritoryCode) and  
                                (d.Add == true or d.Remove == true)))
```

Or, 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. For more information, see “Customizing Differences for New Lines of Business” on page 449 and “Customizing Personal Auto Line of Business” on page 450.

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 further processing. You configure what differences the tree view displays and how the differences appear by customizing difference tree configuration XML files. For more information, see “Difference Tree XML Configuration” on page 441.

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:

- 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:

- 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.

Note: 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. If you have some special need to exclude some properties or entities from difference item lists in all possible cases, see “Policy Difference Overview” on page 431 for details.

PolicyCenter takes the already-generated list of difference items and this XML file and constructs a tree that represents nodes in the tree at appears in the user interface.

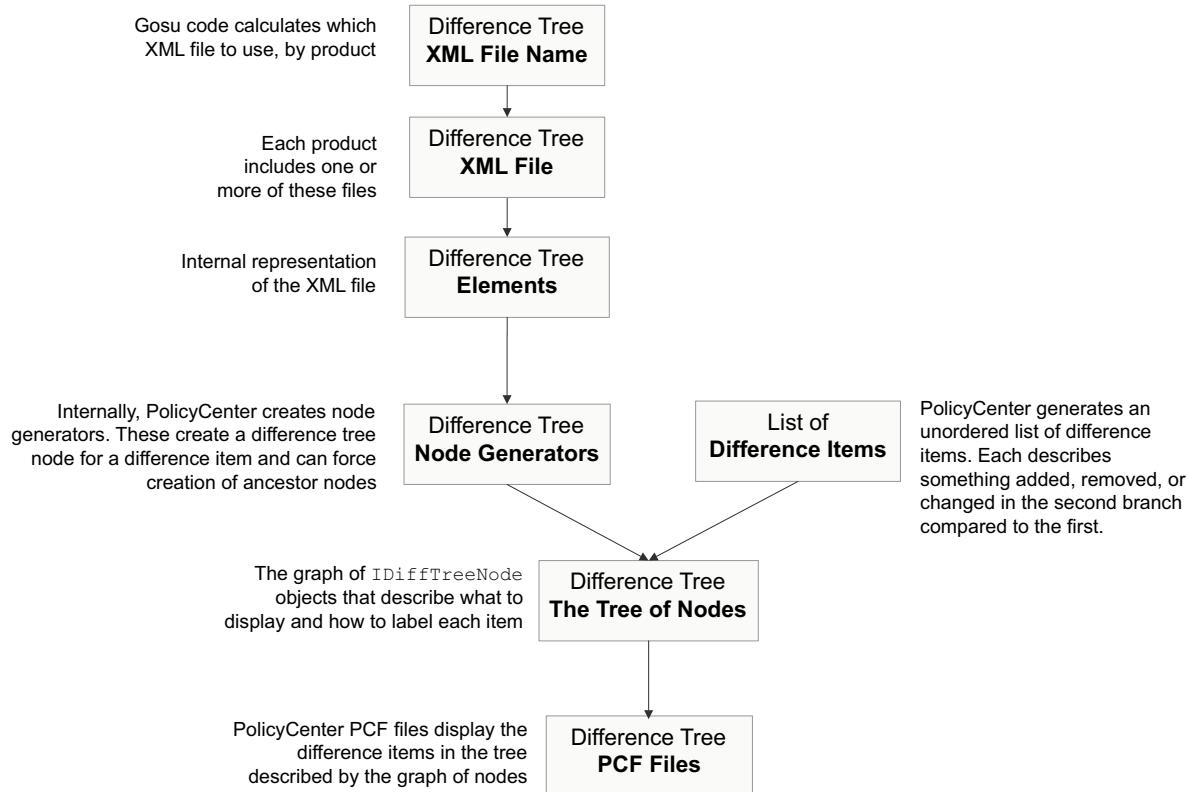
The following steps describe how PolicyCenter uses various objects to create the difference user interface:

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:
 - 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 as follows:
- 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 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 uses this 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.

For example, suppose there is only one difference item representing a change of driver on a car. Your final tree might include multiple nodes that describe the change as well as multiple levels of context of the change:

Root of the line of business → a label that says “Vehicles” → a specific car → the change of driver

The preceding 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.

The number of 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.

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**, and then open **difftree.xsd**.

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)</label>
      <sortorder>3</sortorder>
    </RootEntity>
    <RootEntity type="PolicySecNamedInsured">
      <label>displaykey.Web.Differences.LOB.Common.PolicySecNamedInsured(ENTITY.DisplayName)</label>
      <sortorder>4</sortorder>
    </RootEntity>
    <RootEntity type="PolicyAddlInterest" showadds="false" showremoves="false">
      <label>displaykey.Web.Differences.LOB.Common.PolicyAddlInterest(ENTITY.DisplayName)</label>
      <sortorder>5</sortorder>
    </RootEntity>
    <RootEntity type="PolicyAddress">
      <label>displaykey.Web.Differences.LOB.Common.PolicyAddress(ENTITY.AddressType)</label>
      <sortorder>6</sortorder>
    </RootEntity>
  </Section>
  <Section label="displaykey.Web.Differences.LOB.PA.Drivers">
    <RootEntity type="PolicyDriver">
      <label>displaykey.Web.Differences.LOB.Common.PolicyDriver(ENTITY.DisplayName)</label>
    </RootEntity>
  </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="<![CDATA[<?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>
        </Entity>
      </SubSection>
      <Entity type="VehicleDriver" parentpath="ENTITY.Vehicle">
        <label>displaykey.Web.Differences.LOB.PA.AssignedDriver(ENTITY.DisplayName)</label>
        <sortorder>3</sortorder>
      </Entity>
      <Entity type="PAVhcleAddlInterest" parentpath="ENTITY.PAVehicle">
        <label>displaykey.Web.Differences.LOB.Common.PolicyAddlInterestDetail(ENTITY.DisplayName, ENTITY.AdditionalInterestType)</label>
        <sortorder>4</sortorder>
        <Properties includefromtype="PAVhcleAddlInterest" 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. (Although you can customize the behavior with elements and attributes to omit properties.)

The following subtopics describe the individual elements in the XML format:

- “Difference Tree (Tree Root) Element” on page 446
- “Section Element” on page 446
- “Root Entity Element” on page 446

- “Entity Element” on page 447
- “Subsection Element” on page 447
- “Root Properties Element” on page 447
- “Properties Element” on page 448
- “Property Display Element” on page 448
- “Excluded Property Element” on page 449

Difference Tree (Tree Root) 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`

For example:

```
<Section label="displaykey.Web.Differences.LOB.PA.Drivers">
    <RootEntity type="PolicyDriver"
        label="displaykey.Web.Differences.LOB.Common.PolicyDriver(ENTITY.DisplayName)"/>
</Section>
```

See also

- “Subsection Element” on page 447

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 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

For example:

```
<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.

For example:

```
<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

For example:

```
<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:

- 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. See “Excluded Property Element” on page 449.

For example:

```
<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.

Since 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.

For example:

```
<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
    }
  "/>
</Properties>
```

Property Display Element

The property display element (<PropertyDisplay>) describes the display information for a specific property.

This element has the following attributes:

- **propertyname** – 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 the following (using the first rule that applies):
 - 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 once for each branch that it compares. In other words, if comparing two branches, PolicyCenter calls this once to display Value1 (typically the left column) and once to display Value2 (typically the right column). If you do specify this attribute or set it to null, the default behavior is the following (using the first rule that applies):
 - Use a matching CovTermPattern.DisplayValue display value property if applicable.
 - Booleans display 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 directly into 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.

For example:

```
<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
    }
  "/>
</Properties>
```

Excluded Property Element

The excluded property element (<ExcludedProperty>) excludes a specified property from node generation.

Specify the property to exclude in the propertyname attribute.

For example, the following example shows an excluded property inside a properties element:

```
<Properties includefromtype="IMAccountsReceivable" parentpath="ENTITY" sortorder="2">
  <ExcludedProperty propertyname="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*. For more information about difference helper classes, see “Difference Helper Classes” on page 435. This is discussed further below this bullet list.
 - 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 either:
 - 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
- For details on the format of these files, see “Difference Tree XML Configuration” on page 441. Note the part of that topic that discusses how to tell PolicyCenter where to find your new 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. See “Compare Individual Objects with Matchers” on page 437 for details.

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`. For details, see “Difference Item Subclasses” on page 435.

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. See “Enhancement Reference for Collections and Related Types” on page 261 in the *Gosu Reference Guide*.
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:

```
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:

```
//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 User Interface for Personal Auto

The user interface for the difference tree is controlled by an XML file for each line of business. For details, see “Difference Tree XML Configuration” on page 441. That topic contains the full text of the version for personal auto, the `PADiffTree.xml`, as the example.

APIs for Calculating Differences

In many cases, you only need to think about how the difference calculations are triggered by built-in application logic for the two basic types of differences. However, you can trigger difference calculations by using two APIs: `getDiffItems` and `compareTo`.

Because there are several reasons for calculating differences, you can vary application logic based on the difference reason. Pass a `DiffReason` typecode to the two APIs, with values `TC_INTEGRATION`, `TC_MULTIVERSIONJOB`, `TC_PREEMPTION`, `TC_POLICYREVIEW`, `TC_COMPAREJOBS`. If needed, you can extend the `DiffReason` typelist with additional difference reasons.

Differences Between a Branch at its Based-on Branch

You can get a list of differences between a branch and its based-on branch, which is database generated with `DiffItem` filtering from the policy difference plugin. This is similar to what is done for policy change. Call the `getDiffItems` method on a `PolicyPeriod` entity:

```
myDifferences = myPolicyPeriod.getDiffItems(DiffReason.TC_INTEGRATION)
```

Code similar to this line might be useful for example if you write integration messaging code to keep an external system notified of the changes from Policy Change jobs. You might also use this approach to compare a renewal with the period it was based on. (Although for renewals in some edge cases you might want to use the other approach.)

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 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)
```

Code similar to this might be useful for example if you write new PCF code to compare arbitrary revisions, or perhaps a web service that compares two policy periods.

part VI

Billing, Claim, and Contact Integrations

Billing Integration

PolicyCenter has general support for communication with any billing system through plugin interfaces for outgoing requests and web services for incoming requests. PolicyCenter includes built-in integration with BillingCenter. This topic describes both the general approach for integrating PolicyCenter with a billing system and the details of the built-in BillingCenter integration. Use this topic to learn about strategies for integrating PolicyCenter with billing systems.

This topic includes:

- “Billing Integration Overview” on page 456
- “How Billing Data Flows Between Applications” on page 457
- “Billing Producers and Producer Codes” on page 462
- “Billing Accounts” on page 464
- “Billing Instructions in BillingCenter” on page 467
- “Billing Flow for New-Period Jobs” on page 470
- “Billing Flow for Existing-Period Jobs” on page 473
- “Billing Implications of Midterm Changes” on page 474
- “Billing Implications of Renewals or Rewrites” on page 476
- “Billing Implications for Cancellations and Reinstatements” on page 483
- “Billing Implications of Audits” on page 484
- “Billing Implications for Premium Reporting” on page 486
- “Billing Implications of Delinquency for Failure to Report” on page 487
- “Billing Implications of Deposits” on page 487
- “Implementing the Billing System Plugin” on page 490
- “Implementing the Billing Summary Plugin” on page 500
- “Payment Integration” on page 502
- “Use Integration-Specific Containers for Integration” on page 504

See also

- For more information about web services, see “Web Services Introduction” on page 37.
- For more information about plugins, see “Plugin Overview” on page 123.
- For feature-level and job-level information about PolicyCenter, see “Billing System Integration” on page 733 in the *Application Guide*.
- For the BillingCenter perspective of many of these billing system integration topics, see the *BillingCenter 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
- New accounts
- New producers
- Billing implications of final audits

PolicyCenter requests the following information from billing systems:

- Available payment plans at the time PolicyCenter creates a new policy
- A summary of billing information for a policy that PolicyCenter displays within the PolicyCenter policy file and account file
- A preview of billing information during the submission process
- Availability of agency bill for a producer if the producer has an agency bill plan in the billing system

In contrast with sending and requesting information, 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 PolicyCenter and BillingCenter Integration

The built-in integration between PolicyCenter and BillingCenter uses two mechanisms to connect the applications:

- **Plugins for outgoing requests** – Plugin interfaces define a strict contract between a Guidewire application and additional code that performs a task. Both PolicyCenter and BillingCenter expose plugin interfaces that the application calls to request information from the other system or notify it of new 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. If you use BillingCenter, a built-in plugin implementation implements the billing system plugin. This built-in plugin calls BillingCenter web services, which in response internally create what BillingCenter calls *billing instructions*. If you use a billing system other than BillingCenter, you must write your own billing system plugin implementation to communicate with your billing system as appropriate.
- **Web services for incoming requests** – Both PolicyCenter and BillingCenter expose web services for other applications to query for information or to notify about important actions. If you use a billing system other than BillingCenter, your own billing system plugin implementation communicates with your billing system using some network protocol. The network protocol you use might not necessarily be SOAP but would serve the same purpose.

Integrating Policy Center with Another Billing System

If you want to use a billing system other than BillingCenter, you must create your own implementation class that implements the PolicyCenter billing plugin interfaces. There are two billing plugin interfaces in PolicyCenter, compared in the following table.

Plugin Name	Interface to Implement	Description	PolicyCenter uses during bind, issue, policy change, or renewal
Billing system plugin	IBillingSystemPlugin	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 IBillingSystemPlugin during critical parts of the process of creating or modifying a policy in PolicyCenter. For example, for a new policy the IBillingSystemPlugin 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	IBillingSummaryPlugin	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.

If you want to integrate PolicyCenter with BillingCenter, use and customize the built-in billing system plugin implementations.

See also

- “Asynchronous Communication” on page 459
- “Enabling Integration between BillingCenter and PolicyCenter” on page 107 in the *Installation Guide*
- “Implementing the Billing System Plugin” on page 490
- “Billing System Integration” on page 733 in the *Application Guide*

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, 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 in Policy File in the **Billing** tabs.

In the built-in implementation, to make a midterm billing change, in general you must make changes directly in BillingCenter. However, agents generally cannot log into 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 (getting a bill, sending a check) to direct debit against a credit card or bank account.

These are just the built-in places to view or edit billing information midterm within PolicyCenter. You may add additional PCF pages to show billing data or allow editing. If you want to add pages, follow the examples of these PCF pages. Copy the PCF code and integration code as needed.

For requests that require information to display 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. Synchronous requests are the built-in behavior 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 the built-in asynchronous PolicyCenter to BillingCenter integrations. In other words, create a custom event that sends a message using the event messaging system, and asynchronously send that information to BillingCenter. For more information, see “Asynchronous Communication” on page 459.

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.

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, then sends them to the billing system.

If you use the PolicyCenter built-in integration to BillingCenter, the built-in plugin notifies BillingCenter. In response, BillingCenter internally creates what it calls *billing instructions*, represented by different 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 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 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 built-in 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.

Some PolicyCenter Properties Are Mainly for Billing Systems

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 (especially anything complicated), 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 BillingCenter automatically selects a default value. Typically PolicyCenter does this if there is one fairly likely value.
- Ask the billing system for a list of available plans. PolicyCenter does this because 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 sends 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 154.

Asynchronous Communication

The built-in billing integration 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 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 289

High-level Asynchronous Communication Flow Between PolicyCenter and Billing Center

The following is the high-level flow within PolicyCenter for asynchronous communication to BillingCenter. For example, for a new submission:

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 built-in BillingCenter integration, this class and its related implementation typically implements this behavior. (However, this class does not handle everything. For example, PolicyCenter handles some behaviors 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. Internally, BillingCenter uses these billing instructions to track charges and instructions from the policy system.

See also

- “Billing Instructions in BillingCenter” on page 467.
- “Messaging and Events” on page 289

[Billing Center Sample Code to a Create 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
}
```

Retryability of Messages

If there is an integration error, the message error is always retryable. There are no non-retryable errors in the built-in 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 Applications

There are parts of the PolicyCenter user interface that can directly 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 want 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 URL endpoint URL syntax is different enough from what BillingCenter uses, modify the PCF code for the Exit Point.)

If you want to create additional Exit Points in either application, carefully consider how you want them to work before implementing them. For example, suppose you 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.

Billing Producers and Producer Codes

What BillingCenter calls *producers* and what PolicyCenter calls *organizations* 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 default 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.

PolicyCenter sends messages to the billing system for producers in several 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 default configuration, a PolicyCenter user can change only properties for which PolicyCenter is the system of record.
- Upon system startup, there is special handling of the single internal producer that corresponds to the carrier. The PolicyCenter sends a message to the billing system first to determine if the billing system has the producer for the internal carrier 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 carrier organization.

BillingCenter uses only some properties on producers and producer codes. Specifically, it uses properties related to commission and agency bill plans. Whenever PolicyCenter sets up these entities initially, it sets initial values for those properties 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

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 default in BillingCenter to sending a check every month on the first day of the month.	BillingCenter	n/a	Direct bill commission payment properties
Agency billing plan. PolicyCenter displays a list of agency bill plans retrieved from BillingCenter.	BillingCenter	AgencyBillPlan	AgencyBillPlan

Business property	System of record	PolicyCenter organization entity properties	BillingCenter producer entity properties
Account representatives are optional in BillingCenter. PolicyCenter does not send a value.	BillingCenter	n/a	AccountRep
Unique identifier for the producer within Guidewire applications.	BillingCenter	PublicID	PublicID

If you make a change to the **Contact** in PolicyCenter, then BillingCenter updates the **PrimaryContact**, including properties on subobjects such as address.

Note also that PolicyCenter sends the producer of record to the billing system, not the producer of service, as the producer.

See also

- For complete information about entities, see the *Data Dictionary*.
- For more information about generating and using the *Data Dictionary*, see the “Regenerating the Data Dictionary and Security Dictionary” on page 32 in the *Configuration Guide*.

Producer Codes

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. Each producer has a foreign key to a parent producer code using the **Parent** property on each producer. Producer codes in the hierarchy must belong to the same organization.

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 default PolicyCenter 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 Producer code entity properties	PolicyCenter producer code 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
PolicyCenter displays a list of commission plans retrieved from BillingCenter.	PolicyCenter	CommissionPlan	CommissionPlanID

See also

- For complete information about entities, see the *Data Dictionary*.

Billing Accounts

PolicyCenter shares account information with BillingCenter. There is a one to one mapping between PolicyCenter and BillingCenter accounts. There are some differences in the native structure of the Account entity between the two applications. In PolicyCenter, you generally have a single account for a family, regardless of how the policies in the account are billed. BillingCenter sends invoices from the account. If a family has two policies that they would like to bill separately, the agent creates two accounts in PolicyCenter and PolicyCenter sends over two accounts to BillingCenter. In the built-in integration, you cannot have separate billing accounts for the same policy account. If you want to change this assumption, it would require customizing the existing integration code.

In the built-in integration, PolicyCenter matches the account structure from BillingCenter. So, in this case, PolicyCenter would have two accounts for the family.

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 the account holder
- An addition or update to a billing contact

Some properties on accounts are used only by BillingCenter. These properties are related to billing plan and other account-level settings for invoicing. Whenever PolicyCenter creates these entities initially, it sets initial values for these properties 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.

The following table shows the mapping between account properties in PolicyCenter and BillingCenter. In some cases, properties are set to default values in BillingCenter. 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 serves as the unique ID for mapping between the applications.	PolicyCenter	AccountNumber	AccountNumber
Account name	PolicyCenter	AccountName	AccountHolderContact.Name
Service tier for the account. For more information, see “Service Tier” on page 466.	PolicyCenter	Account.ServiceTier	Account.ServiceTier
Billing plan for the account. BillingCenter has default values. For more information, see “Billing Plan” on page 465.	BillingCenter	BillingPlan	n/a
Frequency. Optional in BillingCenter. Defaults to Every. For more information, see “Invoicing” on page 466.	BillingCenter	Frequency	n/a

Business property	System of record	BillingCenter account entity properties	PolicyCenter account entity properties
Basis. Optional in BillingCenter. Defaults to Month. For more information, see “Invoicing” on page 466.	BillingCenter	Basis	n/a
Optional in BillingCenter. Defaults to true. For more information, see “Invoicing” on page 466.	BillingCenter	FixDueDayOfMonth	n/a
Due day of the month. Optional in BillingCenter. Defaults to 15. For more information, see “Invoicing” on page 466.	BillingCenter	DueDayOfMonth	n/a
Delinquency plan. Defaults in BillingCenter. For more information, see “Delinquency Plan” on page 466.	BillingCenter	DelinquencyPlan	n/a
Invoice delivery types. Optional in BillingCenter. Defaults to Mail.	BillingCenter	InvoiceDeliveryType	n/a
Payment methods. Optional in BillingCenter. Defaults to Responsive.	BillingCenter	PaymentMethod	n/a
Primary payer. Use AccountHolder in policies where the account holder contact references a person. Use BillingContact where the account holder contact references a company.	BillingCenter	Contacts.PrimaryPayer	AccountContact.Roles in two properties: <ul style="list-style-type: none">• AccountHolder• BillingContact
Account holder	PolicyCenter	AccountContact.Roles.Insured	AccountContact.Roles.AccountHolder
Billing contact	BillingCenter	AccountContact.Roles.AccountsPayable	AccountContact.Roles.BillingContact

For complete information about entities, refer to 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 carrier does business and is not usually an option that the producer or underwriter chooses. Therefore, PolicyCenter does not display the billing plan. Whenever BillingCenter creates a new account as a result of the integration, the account gets the default billing plan.

The billing plan can include:

- Thresholds, such as the balance is so low that an invoice is unnecessary
- Whenever payment is due
- How much charge detail to show on the invoice

PolicyCenter does not directly use billing plans or provide a built-in integration for this information.

Delinquency Plan

The delinquency plan determines under which conditions the account becomes delinquent. The plan defines the normal business practices of the carrier and is not usually an option that the producer or underwriter chooses. Therefore, PolicyCenter does not display the delinquency plan. Whenever BillingCenter creates a new account as a result of the integration, the account gets the default delinquency plan.

Invoicing

The frequency and timing of invoicing (due on the 25th of every month, for example) is based on the installment plans of the policies in the account. However, if the account has account-level charges and no policies in the account, then BillingCenter generates an invoice based on `Frequency`, `Basis`, `FixDueDayOfMonth`, and `DueDayOfMonth`. These are optional properties with default values in BillingCenter.

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.

Upon creation of a new account in BillingCenter, the primary payer is determined as follows:

- Since the Account is being created in the context of binding a policy, if there is a `BillingContact` for the policy, then set that contact as the primary payer.
- Otherwise, if there is one `BillingContact` at the account level, set this contact as the primary payer.
- Otherwise, if there is more than one `BillingContact` at the account level, one of them is the primary payer.
- If there is 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:

- Contacts removed from an account in PolicyCenter do not trigger BillingCenter to remove the contact. This avoids removing a contact that the carrier's billing department set up within BillingCenter.
- If you change a contact to a `BillingContact` or `AccountHolder`, then BillingCenter adds that contact to the account.
- If you change the contact role from `BillingContact` or `AccountHolder` to another role in PolicyCenter, then BillingCenter does not update the role. Changing that contact back to a `BillingContact` does not update the role in BillingCenter.

Policy Period Merges

If you merge two accounts in PolicyCenter, PolicyCenter tells BillingCenter to transfer all policy periods from one account to the other account. BillingCenter transfers any unbilled charges also. However, BillingCenter does not transfer the billing history.

Service Tier

Service tiers enable a carrier to provide special handling or value-added services for certain customers, typically high-value customers. When integrated with BillingCenter 8.0 (or later), 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.

For more information about the service tier, see "Service Tier Overview" on page 333 in the *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 built-in integration to BillingCenter, the built-in plugin notifies BillingCenter of these charges. In response, BillingCenter internally creates what it calls *billing instructions*, represented by different 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 using more generalized concepts that roughly correspond to PolicyCenter transaction types.

See also

- For more information about billing instructions, refer to 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 BillingInstruction subtype	Description and important additional 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 a property to specify the deposit requirement that exists after the billing instruction is executed. Also includes an array of <code>PaymentPlanModifier</code> objects to apply to the existing payment plan.
→ <code>BaseGeneral1</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 additional properties
→ Cancellation	<p>Policy cancellation. Includes additional properties:</p> <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 sub-types.
→ 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 carrier 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. Carriers who required specialized policy-period behavior can customize this instruction.
→ PremiumReportDueDate	<p>Informs BillingCenter that a premium report is due on a certain date for a policy. BillingCenter handles delinquency if it does not receive the report on time.</p> <p>Note: In the base configuration, PolicyCenter does not trigger code in BillingCenter that creates this billing instruction.</p>
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 Billing Instruction 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.
cancellation	Cancellation	All cancellations
reinstatement	Reinstatement	All reinstatements
renewal (regular)	Renewal	For regular renewals. Policy must already exist in BillingCenter Note: 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. Note: 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*

Mapping PolicyCenter Financials to BillingCenter Charges

If you use the built-in BillingCenter integration, use the following table as a reference for the mapping of Charge properties between PolicyCenter and BillingCenter.

PolicyCenter property	BillingCenter property	Description
Sum(Transaction.Amount)	Amount	Transaction amount
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.
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.

PolicyCenter property	BillingCenter property	Description
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.
		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.

Whenever PolicyCenter rolls transactions into charges, PolicyCenter sends the following properties:

- Cost.ChargePattern
- Cost.ChargeGroup
- EffectiveDate
- ExpirationDate

Before sending the transactions to the billing system, PolicyCenter sends the transaction property `Transaction.WrittenDate` to the appropriate written date. PolicyCenter calls the policy period plugin method `determineWrittenDate`.

See also

- For details, see “Policy Period Plugin” on page 154.

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 illustration shows the steps and messaging that occur between the applications when you quote a policy for the following jobs: submission, renewal, and rewrite.

PolicyCenter action	Messaging	Billing system action
1. User quotes. 2. User advances to Payment screen.	3. PolicyCenter sends message to get billing options, installment plans, and invoicing plans.	4. Check to see if the producer code of record allows agency bill for this producer. 5. Look up installment and invoicing plans.
6. User selects billing method. 8. User selects installment plan. 9. (Optional) User clicks to preview payments that billing system returned.	6. Billing system returns billing options, available installment plans, and invoicing plans.	
7. User selects invoicing plan. 14. User binds policy.	10. PolicyCenter sends message with the selected payment plan.	11. Calculate payment schedule.
13. User selects invoicing plan.	12. Billing system returns payment schedule which can be viewed in PolicyCenter.	
15. If the billing system does not know about the account, PolicyCenter sends account information. 16. If the user adds an invoicing plan, PolicyCenter sends new invoicing information.	15. If the billing system does not know about the account, PolicyCenter sends account information. 16. If the user adds an invoicing plan, PolicyCenter sends new invoicing information.	17. If account is unknown, create account. 18. If new invoicing plan, add to invoicing plans.
19. 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.	19. 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.	

PolicyCenter action	Messaging	Billing system action
		20. 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 PolicyCenter.
		21. Process and apply the new charges.
		22. Process payment when received.
		23. Call <code>IPolicyPeriod.hasReceivedSufficientPaymentToConfirmPolicyPeriod</code> to determine whether payment is sufficient. Payment may be sufficient, for example, if it is equal to or greater than the amount due on the renewal's first invoice. The call returns true if the payment is sufficient.
24. If <code>PolicyPeriod.ConfirmationNotificationState == NotifyUponSufficientPayment</code> , BillingCenter notifies PolicyCenter that sufficient payment has been received by calling the PolicyCenter <code>PolicyRenewalAPI.confirmTerm</code> method.		
25. Sets <code>Policy.PolicyTerm</code> flag to true.		
26. PolicyCenter sends BillingCenter a message that renewal is confirmed.		27. BillingCenter records the confirmation by setting <code>PolicyPeriod.TermConfirmed</code> 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	<code>PolicyPeriod.Policy.Account.AccountNumber</code>	<code>PolicyPeriod.Policy.Account</code>
Uniquely identify the policy	PolicyCenter	<code>PolicyPeriod.Policy.PublicID</code>	<code>PolicyPeriod.Policy</code>
Policy number	PolicyCenter	<code>PolicyPeriod.PolicyNumber</code>	<code>PolicyPeriod.Policy.PCPublicID</code> (for PolicyCenter use only) <code>PolicyPeriod.PolicyNumber</code> (for external PAS use)
Uniquely identifies the policy period	PolicyCenter	<code>PolicyPeriod.PublicID</code>	<code>PolicyPeriod.TermNumber</code>
Model number	PolicyCenter	<code>PolicyPeriod.ModelNumber</code>	<code>PolicyPeriod.ModNumber</code>
Period start	PolicyCenter	<code>PolicyPeriod.PeriodStart</code>	<code>PolicyPeriod.PolicyPerEffDate</code>

Business property	System of record	PolicyCenter property	BillingCenter property
Period end	PolicyCenter	<code>PolicyPeriod.PeriodEnd</code>	<code>PolicyPeriod.PolicyPerExpirDate</code>
Model date	PolicyCenter	<code>PolicyPeriod.ModelDate</code>	<code>PolicyPeriod.BoundDate</code>
Product (LOB) code	PolicyCenter	<code>Policy.ProductCode</code>	<code>Policy.LOBCode</code>
Assigned risk	PolicyCenter	<code>PolicyPeriod.AssignedRisk</code>	<code>PolicyPeriod.AssignedRisk</code>
Base state	PolicyCenter	<code>PolicyPeriod.BaseState</code>	<code>PolicyPeriod.RiskState</code>
Underwriting company	PolicyCenter	<code>PolicyPeriod.UWCompany.Code</code>	<code>PolicyPeriod.UWCompany</code>
Final audit options	PolicyCenter	<code>PolicyPeriod.FinalAuditOption</code>	<code>PolicyPeriod.UnderAudit</code>
Payment plan	BillingCenter	<code>PolicyPeriod.PaymentPlanID</code>	<code>PolicyPeriod.PaymentPlan</code>
Is this an agency bill policy period	BillingCenter	<code>PolicyPeriod.BillingMethod</code> <code>Value = AgencyBill</code>	<code>PolicyPeriod.AgencyBill</code>
Primary insured	PolicyCenter	<code>PolicyPeriod.PrimaryNamedInsured</code>	<code>PolicyPeriod.PrimaryInsured</code>
Primary producer code	PolicyCenter	<code>PolicyPeriod.ProducerCodeOfRecord</code>	<code>PolicyPeriod.PrimaryPolicyProducerCode</code>

Billing Methods and Payment Plans

The billing method can be agency bill or direct bill. PolicyCenter sends a message to the billing system to see if the producer code of record for the producer allows agency bill in addition to direct bill.

The billing system returns a list of payment plans available for this type of policy and account. The [Payment](#) page displays this information.

The built-in BillingCenter integration supports both of features.

New Periods and Term Confirmed Flag

PolicyCenter can optionally dispatch renewals to the billing system without prior confirmation from the insured. For more information, see “Billing Implications of Renewals or Rewrites” on page 476. 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 built-in billing system plugin automatically 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 built-in 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

- “Policy Renewal Web Services” on page 115.

Billing Flow for Existing-Period Jobs

The following illustration shows the steps that occur between the applications when you quote a policy for job types issuance (not submission), policy change, cancellation, and reinstatement.

PolicyCenter	Messaging	Billing system
1. User quotes in PolicyCenter		
2. User advances to Payment screen.		
Note: This step applies to most jobs, but not Cancellation.		
3. User binds policy.	4. PolicyCenter sends message to update the policy and/or policy period in the billing system. In the same message, PolicyCenter sends billing charges related to that job.	5. Billing system actually process and applies new charges.

Billing Implications of Midterm Changes

In the built-in 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 subtopics list some of the changes that can happen in PolicyCenter. The subtopics do not describe the full set of billing information, some of which must directly change in the billing system in the built-in configuration.

See also

- “Billing Flow for Existing-Period Jobs” on page 473.

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 (<i>for Issuance jobs only</i>)	PolicyCenter	PeriodStart	PolicyPerEffDate
Period end	PolicyCenter	PeriodEnd	PolicyPerExpirDate
Base states	PolicyCenter	BaseState	RiskState
Primary named insured	PolicyCenter	PrimaryNamedInsured	PrimaryInsured
Requires final audit	PolicyCenter	FinalAuditOption typelist: rules, yes, no	RequireFinalAudit Boolean: true or false
Primary producer code	PolicyCenter	Producer	PrimaryPolicyProducerCode

Midterm Changes to Billing Method or Payment Plan

For more information about midterm changes to billing information, see “Billing Integration Overview” on page 456.

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 built-in 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 internally creates an audit billing instruction.
- The policy is reinstated (a reversal of a cancellation).
- The policy is canceled (again) as a flat cancellation. This means a cancellation of 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 PolicyCenter refers to as the *producer of record*. PolicyCenter does not allow midterms changes to the producer of record. Instead, you can add a Producer of Service midterm. By default this new producer becomes the producer of record on renewal. The new producer is the policy’s representative, but there is an extra challenge to 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/rewrite if you want to give commission credit to the new producer. In other words, a cancel/rewrite makes the new producer them the producer of record without waiting for renewal.

PolicyCenter simply sets the producer of service, and a user can change it directly in BillingCenter if you want to give the new producer commissions credit. BillingCenter allows you to allocate commissions between the old and new producer according to some basic rules for splitting the existing charges.

If in PolicyCenter, the Producer of Service changes midterm, PolicyCenter does not push an update to BillingCenter. Whenever Producer of Service becomes Producer of Record on renewal, that producer pushes to BillingCenter with the new policy period (as usual).

If you want to give some commission credit to the new producer midterm, you must go to BillingCenter and make the change there.

If you want to change the producer of record, the easiest way is to issue a cancel/rewrite in PolicyCenter. That sends the new producer to BillingCenter as part of the new policy period.

If you insist on changing the producer of record without a rewrite, you must perform additional customization so that PolicyCenter has the following behavior:

- Allow editing of producer of record in a policy change.
- Send the producer change to BillingCenter.

- Add some method of determining the midterm commission treatment for that change, either a default or some way of asking the underwriter in the PolicyCenter user interface. The built-in implementation does not support this feature.

Note: If the producer of service changes midterm in PolicyCenter, PolicyCenter does not push an update to BillingCenter.

Moving a Policy to a New Account in Midterm

In PolicyCenter, there are two ways to change the account that the policy belongs to. You can:

- 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

This section describes how PolicyCenter handles certain issues related to billing for renewals and rewrites.

Multiple Choices of Renewal Flow

PolicyCenter provides multiple renewal flows from which you can choose. Only choose 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.

Renewal flow	How does PolicyCenter renew a policy?	What happens if the insured pays	What happens 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. Once the status is renewing, whether anything happens immediately depends 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>When the policy renewal issues:</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 do this in the user interface. Within the policy renewal job, click Edit, make changes, click Bind Options, click Renew. The job workflow status becomes renewing.</p> <p>Before PolicyCenter automatically issues the policy, you can manually do this 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. If the insured pays, no special action happens.</p>	<p>By this time, the policy is already issued.</p> <p>If you use BillingCenter, and BillingCenter does not receive a payment, BillingCenter tells PolicyCenter to perform a <i>cancel for non-payment</i>. Whether BillingCenter specifies a flat cancel or midterm cancel is determined by the delinquency plan.</p>

Renewal flow	How does PolicyCenter renew a policy?	What happens if the insured pays	What happens if the insured does not pay
Renewal 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"> There is no built-in 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 payment plan information on a renewal offer letter. The offer must contain a reference number to identify this job. Use the renewal job ID for the PolicyCenter offer identifier. <p>In some cases, the automatic renewal never gets to the renewing status. For example, if there were approval problems. In the policy renewal job, click Quote. The renewal offer status changes to renewing. Because of this status change, your Event Fired rules sends the offer letter to the insured.</p> <p>At this point in time, the property <code>PolicyTerm.Bound</code> is <code>false</code>.</p>	<p>The billing system sends PolicyCenter notice of payment using the PolicyCenter IRenewalAPI 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 <i>suspense payment</i> with an associated offer reference number. A suspense payment is a payment not yet matched to an account and a policy. BillingCenter then calls the PolicyCenter IRenewalAPI web service as mentioned earlier.</p> <p>The PolicyCenter IRenewalAPI web service binds the policy. Binding the policy sets the <code>PolicyTerm.Bound</code> property to <code>true</code>.</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> <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 the billing system receives sufficient payment for the renewal, the billing system notifies PolicyCenter. For example, the billing system calls the PolicyCenter PolicyRenewalAPI web service method <code>confirmTerm</code>. The <code>confirmTerm</code> method does multiple things:</p> <ul style="list-style-type: none"> PolicyCenter sets <code>Policy.PolicyTerm.Bound</code> to <code>true</code>. PolicyCenter sends a message to the billing system that the renewal is confirmed. 	<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	For more information
Bind and cancel	<p>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 must return <code>true</code> from that method. Otherwise, return <code>false</code>.</p>	<p>See “Not Taken Renewals Cancellation” on page 484.</p>

Renewal flow	Configuration overview	For more information
Renewal offer	The policy renewal plugin has a method called <code>isRenewalOffered</code> , which takes a <code>PolicyPeriod</code> object. To support renewal offers, your plugin implementation must return true from that method. Otherwise, return false.	See “Renewal Offers Flows” on page 480. Also see “Not Taken Renewals Cancellation” on page 484.
Confirmed renewal	The policy renewal plugin has a method called <code>doesRenewalRequireConfirmation</code> , which takes a <code>PolicyPeriod</code> object. To support confirmed renewal, your plugin implementation must return true from that method. Otherwise, return false.	See “Confirmed Renewals Flow” on page 481. Also see “Not Taken Renewals Cancellation” on page 484.

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 default configuration, all lines of business use the bind-and-cancel flow.

Renewal Offers Flows

The rest of this subtopic describes how the built-in billing integration handles renewal offers.

PolicyCenter selects the proper payment plan before issuing the renewal, but PolicyCenter does not recalculate premiums (PolicyCenter does not requote the policy period). BillingCenter sends PolicyCenter an amount paid and a payment plan code (optionally) to indicate what the insured chose.

If there is no plan code, then PolicyCenter chooses the payment plan with the largest down payment requirement less than or equal to the amount paid. The idea is to guess whether the insured wants to pay all up-front based on the amount sent in, or to choose 2-pay, 3-pay, and so on. It is best to get an explicit plan choice, but PolicyCenter can choose one that seems most appropriate. PolicyCenter verifies that the amount paid is sufficient to meet the down-payment required. If the person does not send enough to meet the selected plan, then PolicyCenter raises an error and an activity to investigate is generated for a BillingCenter user. If the insured provides no plan code, then PolicyCenter checks whether the amount is sufficient for any plan. If no (adequate) payment signal is received in time, then 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 using the `IPolicyRenewalAPI` 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 carrier bind the policy.

The carrier initially sends a renewal notice (including pricing and payment plans) but does not book premiums for accounting or statistics. If payment is received, then the billing system tells PolicyCenter to bind. Otherwise, a built-in 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. See “Not Taken Renewals Cancellation” on page 484.

In some cases, a carrier 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 carrier 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 carriers may accept payment within a grace period. In other words, if the payment is past the grace period, then 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 built-in 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), then 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

Customize When to Use Renewal Offers

As mentioned in the table earlier in this topic, all line of businesses use the bind-and-cancel, except for business auto that uses renewal offers. For built-in lines of business, you can change which flow to use. For new lines of business you create, you can choose either renewal flow.

PolicyCenter configures the renewal flow choice by asking the registered implementation of the policy renewal plugin interface (`IPolicyRenewalPlugin`). PolicyCenter calls the plugin's `isRenewalOffered` method. If this method returns true, PolicyCenter uses a renewal offer for this policy renewal, otherwise it uses the bind-and-cancel flow.

The built-in implementation of this plugin looks like:

```
override function isRenewalOffered( periodToRenew : PolicyPeriod ) : boolean {  
    return periodToRenew.Job.type is Renewal  
        and periodToRenew.Policy.ProductCode == "BusinessAuto"
```

If you override this plugin, check the line of business using the pattern shown in this example within your `isRenewalOffered` method. Or, if you use the built-in implementation of this plugin, just customize the method directly. 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 carriers to bind a policy before payment.

See also

- “Multiple Choices of Renewal Flow” on page 477
- “Renewal Offers Flows” on page 480
- “Confirmed Renewals Flow” on page 481

Confirmed Renewals Flow

For some carriers, neither the bind-and-cancel nor the renewal offers flows are appropriate.

Bind-and-cancel renewals have one main limitation:

- If the insured fails to pay, then there is no coverage provided and it 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 carrier must make it clear to the insured that coverage is uncertain. If the carrier receives payment within the deadline, then there is coverage, otherwise the carrier must deny the claim. This is especially a problem if carriers 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 carrier 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 carriers want BillingCenter in charge of the renewal offers, particularly for receiving electronic payments.

Confirmed renewals are an alternative flow that combines some of the best qualities of bind-and-cancel and renewal offer flows:

- Confirmed renewals allows the job to complete immediately, like the bind-and-cancel flow.
- BillingCenter is in 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 default configuration, for BillingCenter users, PolicyCenter sends this information in the billing instruction `TermConfirmed` property. For more information, see “Billing Flow for New-Period Jobs” on page 470.
- Some carriers do not want to book written and earned premiums or billed receivables until a renewal is paid by the insured. In confirmed renewal, the carrier can confirm that the insured actually wants the policy before starting those processes. In other words, you can wait until confirmation before accruing written and earned premium. However, in the default configuration of PolicyCenter there is no waiting until confirmation. You can change this built-in behavior. In contrast, the bind-and-cancel approach starts immediately booking financials because there is no way to encode the concept of pending confirmation.

Note: The implications for premiums and receivables with confirmed renewals varies widely by carrier. Carefully consider how you want confirmed renewals to work before you implement them.

- For confirmed renewals, the billing system gives PolicyCenter positive confirmation whenever the insured pays the billing system. This is not true of the bind-and-cancel flow.
- For confirmed renewals, PolicyCenter knows that the insured never confirmed the policy and therefore the policy is never legally binding. This is useful information for handling ambiguities like those mentioned earlier in this topic. This is not true of the bind-and-cancel flow.

See also

- “Multiple Choices of Renewal Flow” on page 477
- “Renewal Offers Flows” on page 480
- “Confirmed Renewals Flow” on page 481

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 115

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, then PolicyCenter retrieves the latest value from BillingCenter at the time it starts the renewal or rewrite. These properties are:

- Agency Bill or Direct Bill (only if there is a choice)
- Payment Plan
- For other properties, BillingCenter copies those values to the new periods from the prior period.

Billing Implications for Cancellations and Reinstatements

This topic discusses billing issues related to cancellations and reinstatements. For the high-level flow for this type of job, see “Billing Flow for Existing-Period Jobs” on page 473.

See also

- “Billing Implications for Premium Reporting” on page 486

Cancellations That Start in PolicyCenter

If PolicyCenter binds a cancellation, PolicyCenter notifies BillingCenter, which internally creates a cancellation billing instruction to handle it.

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 carrier 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 carrier that do not want the renewal.

Cancellations That Start in BillingCenter

Several types of cancellations start in BillingCenter:

- “Delinquency Cancellation” on page 483
- “Not Taken Renewals Cancellation” on page 484

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 default configuration, PolicyCenter calculates the actual cancellation date using the minimum lead time required by law.

Whenever 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 at the time it receives the Cancellation billing instruction.

IMPORTANT If delinquency triggers cancellation, BillingCenter requests immediate cancellation. The built-in 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 the differences between the types of renewal flows, see “Multiple Choices of Renewal Flow” on page 477.

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 initiates a flat cancel order in PolicyCenter using web services. In this case, the cancellation reason is *not taken* rather than *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.

Note: 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*.

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

- “Billing Implications of Renewals or Rewrites” on page 476

Billing Implications of Audits

For the high-level flow for this type of job, see “Billing Flow for Existing-Period Jobs” on page 473.

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. At the time 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. A carrier 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 built-in 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

There are several things that a policyholder must understand in a bill for an audit:

- The total audited premium for the period. This is basically a quote page from PolicyCenter.
- The amount due on their invoice

This might seem straightforward, but carriers traditionally provide a bill showing the value of total amount due, using the following formula:

$$\text{TOTAL_AMOUNT_DUE} = \text{TOTAL_AUDIT_PREMIUM} - \text{AMOUNT_PREVIOUSLY_BILLED_AND_COLLECTED}$$

PolicyCenter does not know the total amount due because PolicyCenter does not know what the billing system already collected.

However:

- PolicyCenter knows the difference between the (1) audited premium and (2) what was previously sent to billing (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 carrier 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.

A carrier 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 built-in BillingCenter integration uses the second approach. The built-in 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, then other changes may be made to the policy, 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 built-in BillingCenter integration creates another audit request with additional charges. BillingCenter internally creates an audit billing instruction to handle this.

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

For the high-level flow for this type of job, see “Billing Flow for Existing-Period Jobs” on page 473.

Choosing a Reporting Plan

PolicyCenter handles some policies on a reporting basis. In this case, from the perspective of a PolicyCenter user, you choose a reporting plan instead of an installment payment plan.

If you use BillingCenter, the reporting plan still implies a simple payment plan in BillingCenter.

Sending Reported Premiums to Your Billing System

For a policy on 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 built-in 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 BillingCenter to what it calls a Premium Report billing instruction.

In addition to sending the charges, PolicyCenter tells the billing system 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 carrier 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 built-in integration to BillingCenter includes information about whether the payment was received. Internally, 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 sent together and the carrier usually deposits the check immediately before the report arrives in the audit department for data entry. If so, then the billing system assigns the payment to the account as a suspense payment and waits for the report to come through.

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.

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, therefore PolicyCenter tracks and initiates delinquency for premium reporting. In the built-in integration, PolicyCenter simply 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 collateral collected up front on a policy that bills based on reporting. It is different from a *down payment*, which is an initial payment of a portion of the premium. In the insurance industry, informally people use both terms loosely. Guidewire documentation uses these terms with their precise meanings.

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.

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 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.

PolicyCenter determines the necessary deposit amount for each policy period. PolicyCenter sends that amount to BillingCenter as a number (not an incremental amount) for each PolicyCenter job that PolicyCenter sends to BillingCenter.

BillingCenter tracks this amount on a per policy period basis and uses it to establish a required deposit, 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. However, you can configure BillingCenter to support a rolling deposit (max of all open periods) or a straight deposit for each period (sum 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.

Details by Job Type

Implications for deposits vary by job type:

- “Deposits in Submission, Renewal, or Rewrite Jobs” on page 488
- “Deposits in Policy Change and Reinstatement Jobs” on page 489
- “Deposits in Premium Reports” on page 489
- “Deposits in Cancellation” on page 489
- “Deposits in Final Audit” on page 490

Deposits in Submission, Renewal, or Rewrite Jobs

Upon selection of a Reporting Plan on the Payments Page, PolicyCenter displays a default deposit requirement. PolicyCenter gets this default from the audit schedule that you configure in Studio.

The deposit amount is a percent of the total cost subject to reporting. Typically, taxes are not subject to reporting. The subject-to-reporting premium is actually a subset of the total premium in 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 it in the user interface.

Upon bind, PolicyCenter sends the deposit collateral amount to BillingCenter with the `PolicyPeriod`. In response, BillingCenter internally stores this data in the property

`PolyBillingInstruction.DepositRequirement`. This deposit is the amount of collateral to consider segregated for the specified policy period (this policy term).

At the time 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 `CollateralRequirement.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, then 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.

However, at the time the policy renews, carriers 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. There is overlap of the renewal period if it binds in advance of 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 carriers 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 (from audit, for example) for the old period and retains the deposit for the renewal period.

Typically the carrier 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.

At the time 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 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 adjusted as premiums increase or decrease from period to period. However, the deposit is not returned at the end of a period unless the policy is canceled or non-renewed.

The built-in BillingCenter integration only handles the first option, 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. The user has the ability to override the deposit percentage. At the time the job binds, PolicyCenter resends the full deposit amount to BillingCenter. This amount is the total value required for the policy period, not the change in the deposit value.

At the time 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 at the time premium report transactions complete. There is no change to the deposit value during these transactions.

Deposits in Cancellation

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, at the time 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 as the new collateral requirement.

In BillingCenter, the active collateral requirement status changes to *closed*. Money paid and held as *cash held by requirements* balance releases, which sets the money to the category of collateral cash held. This means that the collateral cash is available for transfer. Typically the carrier 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.

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 applies to any final audit, both expiration and cancellation.

In BillingCenter, the active collateral requirement changes to *closed*. Money paid and held as a *cash held by requirements* balance releases to the category of collateral cash held. This means that the collateral cash is available for transfer. Typically the carrier 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.

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. This results in reversing the final audit, followed by a policy change, followed by scheduling a new final audit. In such a case, a carrier 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 built-in 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 with each type of message from PolicyCenter to BillingCenter. After a deposit is released, only a reinstatement unreleases the deposit.

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. For a comparison of the billing system plugin and the billing summary plugin, see “Billing Integration Overview” on page 456.

Note: If you use BillingCenter, the built-in plugin implementation handles these tasks. Skip this topic altogether, and see “Enabling Integration between BillingCenter and PolicyCenter” on page 107 in the *Installation Guide*.

Your plugin implementation must implement all the methods defined by the plugin interface. This plugin handles a wide variety of billing-related tasks:

- “Account Management” on page 491
- “Policy Period Management” on page 492
- “Producer Management” on page 496
- “Billing System Notifications of PolicyCenter Policy Actions” on page 494
- “Agency Bill Plan Availability Retrieval” on page 497
- “Commission Plan Management” on page 498
- “Installment Previews” on page 499
- “Payment Plans and Installment Previews” on page 498
- “Updating Contacts” on page 499

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 causes a messaging event to fire.
 - This might happen because of a custom messaging event that some code triggers. For example, completing a policy change.
 - This might happen due to automatic messaging events that trigger because of changes in Guidewire entities (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

- “Asynchronous Communication” on page 459.

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 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 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 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 has 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 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 it.

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 simply 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. For more information about `issuePolicyChange`, see “Billing System Notifications of PolicyCenter Policy Actions” on page 494.

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.

There are other types of cancellations that the billing system starts. 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. For more information, see “Policy Cancellation and Reinstatement Web Services” on page 105.

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) 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.

It is important to understand that PolicyCenter calls `cancelPolicyPeriod` for the actual cancellation. Cancellation can often be a complex process typically with legally-required notification delays. PolicyCenter does not call `cancelPolicyPeriod` at the time of a request to initiate cancellation. Instead, 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.

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.

Get Period Information from Billing System

Your plugin must also 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 notably 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.

- In Gosu, the fully-qualified class name of this class is `gw.api.billing.PolicyPeriodBillingInfo`
- In Java, the fully-qualified class name of this class is
`external.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

There are various user actions in PolicyCenter that might 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 simply populate properties in an object that represents the policy period and recent changes. Then, send that request to your billing system in a web service request. This is what the built-in billing system plugin does to notify Guidewire BillingCenter.

If you do not use BillingCenter, you must handle all of the events listed in the following table. This table includes the following information:

- A PolicyCenter action.
- The corresponding `IBillingSystemPlugin` method that PolicyCenter calls for this action.
- A description of the action.

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.

For each of the following actions, you probably need to send the following basic policy period data:

- `period.PeriodId.Value`
- `period.PolicyNumber`
- `period.EditEffectiveDate.toCalendar()`
- `ChargeInfoUtil.getChargeInfos(period)`

For some actions, the following table lists additional properties you might want to send to your billing system and other important information about each method.

Action	Method name	Description
policy change	issuePolicyChange	<p>Issue a policy change in the billing system. It is important to understand that PolicyCenter does not notify the billing system about all changes to the policy. Specifically, PolicyCenter notifies the billing system about the following policy changes:</p> <ul style="list-style-type: none"> • Premium changes • Base state changes • Primary insured changes <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"> • period.PolicyChange.Description • period.BaseState.Code • period.PeriodStart.toCalendar() • period.PeriodEnd.toCalendar() • period.PrimaryNamedInsured.AccountContactRole.AccountContact.Contact (a Contact entity)
issuance	issuePolicyChange	<p>PolicyCenter also calls the issuePolicyChange method for Issuance as well as policy changes. It is important to understand that PolicyCenter does not call this as part of the bind process, but calls it for issuance if it happens as a separate action. If the user performs just a bind, PolicyCenter simply creates the new PolicyPeriod locally and then calls the billing system createPolicyPeriod method (see “New Policy Periods” on page 492). If the user performs a bind and issue as one user interface action, PolicyCenter calls createPolicyPeriod only. If the user performs bind only, then PolicyCenter calls createPolicyPeriod only. Any issuance request that occurs later results in PolicyCenter calling issuePolicyChange.</p> <p>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 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"> • period.Audit.AuditInformation • period.Audit.AuditInformation.AuditPeriodEndDate.toCalendar() • period.Audit.AuditInformation.AuditPeriodStartDate.toCalendar() • period.Audit.PaymentReceived
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>
final audit	issueFinalAudit	<p>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.</p>

Action	Method name	Description
schedule final audit	scheduleFinalAudit	<p>Schedule a final audit on a policy period in your 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"> • period.PolicyNumber • period.PeriodId.Value
waive final audit	waiveFinalAudit	<p>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.</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"> • period.PolicyNumber • period.PeriodId.Value

Producer Management

PolicyCenter tracks various types of organizations with the `Organization` entity. Every organization has an organization type in its `OrganizationType` property, which holds a typecode. If the typecode of the organization type has the category (the `Category` property) called `Producer`, PolicyCenter considers this an external producer that has a corresponding entity 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 *producers*.

Producers

To accommodate new producers, implement the `createProducer` method, which takes as arguments the `Organization` entity 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 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 simple method `producerExists`. It 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 also 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

Similarly, you must implement the `updateProducerCode` method, which takes the same arguments but returns nothing.

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

You also must implement the method `getAvailableBillingMethods`. It 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.

- In Gosu, the fully-qualified class name is `gw.api.billing.AgencyBillPlanSummary`
- In Java, the fully-qualified class name is `external.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.

- In Gosu, the fully-qualified class name is `gw.api.billing.CommissionPlanSummary`
- In Java, the fully-qualified class name is `external.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

Payment Plans

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 as part of the user interface. 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 (submission, renewal, rewrite).

Implement the following plugin methods:

- `retrieveIssuancePaymentPlans` – Retrieves payment plans for a new submission
- `retrieveRenewalPaymentPlans` – Retrieves payment plans for a renewal
- `retrieveRewritePaymentPlans` – Retrieves payment plans for a policy rewrite

All three methods take a `PolicyPeriod` entity and return an array of `PaymentPlanSummary` entities.

Set the following properties on each `PaymentPlanSummary` entity:

- **BillingId** – The ID of the plan.
- **Name** – The name of the plan.
- **PaymentCode** – The payment code of the plan.
- **DownPayment** – The down payment of the plan. This is the largest installment. The amount must be a `nonnegativemoney` value.
- **Installment** – The installment payment of the plan. This is the sum of the down payment and all installments combined. The amount must be a `nonnegativemoney` value.
- **Total** – The total payment of the plan. The amount must be a `nonnegativemoney` value.
- **Notes** – Notes related to the plan.
- **PolicyPeriod** – A foreign key reference to the related `PolicyPeriod` entity.

These methods get the summary of the allowed plans.

Installment Previews

Similar to the earlier discussion about 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 after a user selects a payment plan to show the user what the installments to expect. It is important to understand that by the time PolicyCenter calls your plugin, the payment plan is already set.

Note: PolicyCenter always calls this method synchronously with respect to the user interface.

Implement the following plugin method:

- `retrieveInstallmentsPlanPreview` – Retrieves previews of installments for a new policy, renewal, or policy rewrite job. This method takes a `PolicyPeriod` entity and return an array of `PaymentPreviewItem` objects.

Note: PolicyCenter always calls this method synchronously with respect to the user interface.

This method takes a `PolicyPeriod` entity and returns an array of `PaymentPreviewItem` objects.

- In Gosu, the fully-qualified class name of this class is `gw.api.billing.PaymentPreviewItem`
- In Java, the fully-qualified class name of this class is `external.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 `BigDecimal` value

Note: The methods described in this section get installment previews, not actual payment plans on a policy period. To get actual payment plans, see “Payment Plans” on page 498.

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 a 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. The parameters to the method are a `Contact` entity 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 built-in implementation, PolicyCenter does not notify the billing system that the contact no longer has that role.

Other Plugin Methods

There are other methods on the billing plugin interface:

- `addPaymentInstrumentTo` – Adds a payment instrument to an account. and return 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.
- `searchForAccounts` – Search for accounts in the billing system using a search criteria.
- `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.

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.

For a comparison of the billing system plugin and the billing summary plugin, see “Billing Integration Overview” on page 456.

If you use BillingCenter as your billing system, you can skip this topic. Just use the built-in plugin implementation as described in the “Enabling Integration between BillingCenter and PolicyCenter” on page 107 in the *Installation Guide*.

Java Interfaces Used by the Billing Summary Plugin

The methods in the billing summary plugin interface return instances of classes that implement one of several Java interfaces. You must define your own concrete implementations of these Java interfaces.

The Java interfaces are in the `gw.plugin.billing` package. For the billing summary plugin you must define concrete implementation for the following interfaces:

- `BillingAccountInfo`
- `BillingInvoiceInfo`
- `BillingPeriodInfo`

The Java interfaces define methods, primarily property getter methods. In Gosu, these getters are exposed as properties. For example, the `BillingPeriodInfo` Java interface has a `getPeriods` method. In your Gosu concrete implementation of this interface, you implement a getter for `Periods` property. Gosu code can access the `Periods` property directly or by calling the `getPeriods` method. See “Java get/set/is Methods Convert to Gosu Properties” on page 125 in the *Gosu Reference Guide*. You can also add additional getters and setters for other properties that you create.

PolicyCenter includes two implementations of `IBillingSummaryPlugin`:

- `BCBillingSummaryPlugin` – Built-in plugin implementation for PolicyCenter integration with BillingCenter.
- `StandAloneBillingSummaryPlugin` – Example plugin implementation that mimics integration with a third-party billing system.

Examine these classes to see ways of implementing the plugin and its interfaces.

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

From the **Payments** page 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.

IMPORTANT External payment systems 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.

For demo purposes, the default configuration calls a demo version of a payment system that PolicyCenter itself implements with PCF pages. You can implement something similar to this system if you want to directly collect the payment source information and use it to initiate the payments. In such an approach, you would not use tokens. You would directly contact the payment processor with the payment source information you already collected and stored.

WARNING Check with your legal department about all regulatory requirements relating to credit card information before going to production.

The general flow is as follows:

1. Start a submission, renewal, or rewrite job.
 2. After quoting but before binding the policy, PolicyCenter displays a payment screen.
 3. PolicyCenter displays billing methods, payment methods, and associated installment or reporting plans retrieved from the billing system. After you select a payment method, you can preview payments retrieved from the billing system. This part of the wizard is similar for all lines of business.
- In the Payments page, the part of the page that relates to payments uses the PCF file `BillingInvoiceStreamInputSet.pcf`. This PCF file lists payment-related information and provides an option to add payment information for recurring payments. The default user interface collects recurring payments only, not down payments.
4. In `BillingInvoiceStreamInputSet`, there is an **Add** button. Click the **Add** button.
 5. The **Add** button calls a Gosu method called `externalPaymentLocation`. That method exists within the `Code` tab of the `BillingInvoiceStreamInputSet` PCF page.
 6. The `externalPaymentLocation` method calls a PCF exit point to navigate to the payment system user interface.

Typically, the payment system would be on a separate physical computer across the Internet.

For demo purposes, the default configuration of this function calls a demo version of a payment system that PolicyCenter itself implements.

For a production environment, you must change the code for `externalPaymentLocation` method to contact your real payment system. For details, see "Configuring PolicyCenter to use a Real Payment System" on page 503

7. One of the last steps in `externalPaymentLocation` is to invoke the PCF exit point called `CreatePaymentInstrument`.

In the default (demo) configuration:

- a. PolicyCenter calls the `CreatePaymentInstrument` exit point. This is the standard exit point for a real payment system
- b. PolicyCenter calls the `NewPaymentInstrument` entrypoint to display the demo payment collection page. This entry point is for the demo payment system only, not for use in production.
- c. After you enter a payment, that demo payment page uses the PCF exit point called `FinishPaymentInstrument`. This exit point is for the demo payment system only, not for use in production.
- d. That exit point then calls the PolicyCenter PCF entry point called `JobWithNewPaymentInstrument`. This is the standard entry point that a real payment system would use as a PolicyCenter entrypoint.

In a real implementation

- a. `CreatePaymentInstrument` calls the payment system uses the external payment system exit point.
- b. You enter a new payment in the external payment system.
- c. After it completes, the payment system navigates to PolicyCenter using the PolicyCenter PCF entry point `JobWithNewPaymentInstrument`.

8. PolicyCenter notifies the billing system of the new payment.

PolicyCenter gets the registered billing system plugin (`IBillingSystemPlugin`) implementation and calls its `addPaymentInstrumentTo` and `getExistingPaymentInstruments` methods.

For details of fields and behavior of the Payments page, see “Payments” on page 746 in the *Application Guide*.

Configuring PolicyCenter to use a Real Payment System

For demo purposes only, in the default configuration this function calls a demo version of a payment system that PolicyCenter itself implements.

To integrate with a real payment system, you must make two changes to PolicyCenter.

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 `externalPaymentLocation` method to contact your real payment system.

There is a line that looks like the following code. Note the commented out URL

```
var returnUrl = gw.api.system.PCConfigParameters.PaymentSystemURL.Value  
// gw.plugin.Plugins.get(gw.plugin.webconfig.IPolicyCenterWebConfigPlugin).PolicyCenterURL
```

Change the assignment instead to use the commented-out code:

```
var returnUrl =gw.plugin.Plugins.get(gw.plugin.webconfig.IPolicyCenterWebConfigPlugin).PolicyCenterURL
```

3. In the demo configuration, only the payment method (type of credit card) is sent to PolicyCenter. Configure the entry point `JobWithNewPaymentInstrument` to have additional attributes that the payment system sends to PolicyCenter.

Use Integration-Specific Containers for Integration

The built-in implementation of the integration passes information from PolicyCenter to BillingCenter, and from BillingCenter to PolicyCenter. These applications use web services defined in Gosu to pass information to each other. However, the objects passed between applications are not direct references to persisted Guidewire entities. Each application defines the core APIs to pass across containers that encapsulate important information. Many of these objects have the suffix `Info` or the prefix `External`.

If you modify the integration, Guidewire strongly recommends 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.



chapter 23

Multicurrency Integration between BillingCenter and PolicyCenter

BillingCenter and PolicyCenter support multicurrency integration with each other in their default configurations. However, you might need to modify integration code in your configurations of BillingCenter and PolicyCenter to enable the integration appropriately.

This topic includes:

- “Set up Currencies for Multicurrency Integration” on page 505
- “Configure Account Numbers for Multicurrency Accounts in BillingCenter” on page 506
- “” on page 506

See also

- “Multicurrency Integration Between BillingCenter and PolicyCenter” on page 748 in the *Application Guide*
- “Enabling Multicurrency Integration” on page 608 in the *Configuration Guide*

Set up Currencies for Multicurrency Integration

In a multicurrency InsuranceSuite integration between BillingCenter and PolicyCenter, you must set the default application currency the same in both applications. Using the same default application currency in each core application of Guidewire InsuranceSuite always is required. In addition, you must configure each application with the currencies that you want them to share for billing purposes.

For example, In PolicyCenter you allow policies to be settled in U.S. dollar, European Union euro, and Japanese yen. For European policies, you allow assets to be valued in European Union euro and British pound. In PolicyCenter you set up pounds for use as a coverage currency, and you set up dollar, euro, and yen for use as settlement currencies.

In BillingCenter, you also set up U.S. dollar, European Union euro, and Japanese yen as currencies. BillingCenter and PolicyCenter share these currencies for billing and settlement purposes. However, you do not setup British pound as a currency in BillingCenter. In the base configuration of PolicyCenter, each policy can have only one settlement currency. So, PolicyCenter converts pounds to euro before it calculates premiums and other charges on a multicurrency policy. PolicyCenter sends the charges to BillingCenter in euro.

See also

- “Configuring Currencies” on page 113 in the *Globalization Guide*

Configure Account Numbers for Multicurrency Accounts in BillingCenter

In PolicyCenter, for an account with policies in more than one currency, only a single account number is visible. However, in BillingCenter there is a set of affiliated accounts corresponding to each currency used by the original PolicyCenter account. BillingCenter does not generate special account numbers for affiliated accounts. In the base configuration, BillingCenter gives each account a unique and unrelated number, regardless of whether the account is an affiliated currency account or not.

Affiliated Account Numbers and Billing Communication

The insured can receive invoices or other correspondence that references the account number for an affiliated currency specific BillingCenter account. You can configure BillingCenter to assign an account number to affiliated accounts that relates in an obvious way to the original account number.

For example, a policyholder with policies in multiple currencies calls to inquire about an invoice that references an affiliated account number. It will help the billing clerk if the account number for that affiliated account is clearly related to the original account number.

Assign Account Numbers to Affiliated Multicurrency Accounts

When BillingCenter creates an affiliated currency account you can replace the automatically generated account number. Create a custom account number that include the principal account number visible in PolicyCenter. Then assign this custom account number to the affiliated account.

BillingCenter creates affiliated accounts in the `createAccountForCurrency` method in the Billing web service API. You can configure this method to create a specific and meaningful account number for affiliated accounts. Use the `parentAccount` parameter in the `createAccountForCurrency` method to access the principal account number.

For example, to assign a similar account number to BillingCenter affiliated accounts, you can take the principal account number and append the currency of the affiliated account. Make this change in the `createAccountForCurrency` method. After the affiliated account is created, you can insert code similar to the following:

```
account.AccountNumber = parentAccount.AccountNumber + "." + currency.DisplayName
```

Claim and Policy Integration

This topic describes web services, plugin interfaces, and tools for communicating with claim systems such as Guidewire ClaimCenter.

This topic includes:

- “Claim Search from PolicyCenter” on page 507
- “Policy System Notifications” on page 509
- “Policy Search Web Service (For Claim System Integration)” on page 509
- “PolicyCenter Exit Points to ClaimCenter and BillingCenter” on page 511
- “PolicyCenter Product Model Import into ClaimCenter” on page 512
- “Policy Location Search API” on page 523

See also

- “Web Services Introduction” on page 37
- “Plugin Overview” on page 123

Claim Search from PolicyCenter

PolicyCenter calls the claim search plugin (`IClaimSearchPlugin`) to search for claims against a policy. You can implement this plugin interface to use PolicyCenter with an existing claim management system.

PolicyCenter includes a built-in claim search plugin implementation that communicates with Guidewire ClaimCenter. However, you can implement this plugin interface to use PolicyCenter with another claim management system.

This plugin has the main method, `searchForClaims`, which takes a `ClaimSearchCriteria` object, which contains all the properties a PolicyCenter user is searching for. This includes properties such as:

- `Account` - the account
- `DateCriteria` - the date criteria
- `Policy` - the policy entity (check for important properties)

- **PolicyNumber** - the policy number
- **PublicID** - the public ID or primary key in the remote system

View the JavaDoc or *Data Dictionary* for more details of this object.

Your plugin must look up the claims for the dates in the **DateCriteria** object.

This method returns a claim set (**ClaimSet**) object, which encapsulate a list of zero, one, or more claims. Create your array of claims and store the array in the claim set property **Claims**. Carefully consult the Data Dictionary to determine the required properties for claims.

In addition to the built-in implementation that calls out to ClaimCenter, PolicyCenter includes a demo implementation of the plugin that does not actually call out to anything.

The built-in implementation that calls out to ClaimCenter might be instructive on 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)

        for (pcClaim in claimResult) {
            var claim = addClaimToClaimSet(pcClaim, result)
            mapClaimToPeriod(claim, pcClaim.PolicyNumber)
        }

        var claimFilter = new ClaimPolicyPeriodFilterSet(result.Cclaims)
        result.setClaimsFilter(claimFilter)
    })
}

return result
}
```

Your version of this plugin must actually call out to the claim system and generate a new claim set and a series of new claims as shown. 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 a PolicyCenter wants more information about a claim, PolicyCenter asks your claim search plugin to retrieve it. PolicyCenter calls the **getClaimDetailByClaimNumber** method for this task. This method takes an entire **Claim** entity, but you might only need the claim number property from that entity to send to the external system. It returns a **ClaimDetail** object, which contains a small amount of details about the claim in its properties. Refer to the Javadoc 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 includes a general architecture for notifications from claim systems to PolicyCenter. The only built-in notification type in the default implementation is to detect large losses.

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.

IMPORTANT The definitions of the thresholds for what counts as a large loss are defined from within the claim system, not in PolicyCenter.

The PolicyCenter implementation of the notification web service does the following:

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 Details](#)

The API has only a single method, called `claimExceedsThreshold`. It takes three arguments:

- Loss date, as a `java.util.Calendar` object
- Policy number, as a `String`
- Gross total of money, as a `MonetaryAmount` value.

The `claimExceedsThreshold` method returns no value.

[Policy Search Web Service \(For Claim System Integration\)](#)

If you use a supported version of ClaimCenter, ClaimCenter uses the `CCPolicySearchIntegration` web service to retrieve policy summaries from PolicyCenter based on search criteria. Typically you do not need to call this directly, since ClaimCenter calls this automatically 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 specifically 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 wants 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 desired 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( 2006, 8, 21 );
criteria.AsOfDate = calendar.getTime;

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. Refer to the SOAP API Javadoc or the data model for more information about the properties on `CCPolicySummary` and `CCPCSearchCriteria`.

Given the returned list of policy summaries, the claim system user might select one of these to get and associated with a claim.

Retrieving a Policy

After the claim system knows what policy it wants, the claim system calls the `retrievePolicy` method of this web service to retrieve the policy.

The claim system identifies the requested policy with the following:

- Policy number as a `String`, which corresponds to the `CCPolicySummary` field `_policyNumber`.
- Effective date as a `java.util.Calendar` object, which corresponds to the `CCPolicySummary` field `_effDate`

For example:

```
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. Refer to the SOAP API Javadoc or the *Data Dictionary* for more information about the properties on `Envelope`. Be sure to note which fields are required.

IMPORTANT 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 a carrier to provide special handling or value-added services for certain customers, typically high-value customers. When integrated with ClaimCenter 8.0 (or later), 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. For more information about the service tier, see “Service Tier Overview” on page 333 in the *Application Guide*.

Extending 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 typecode in PolicyCenter exceeds 50 characters, it is trimmed before sending to claim system in a policy search.

PolicyCenter implements this using the `ProductModelTypeListGenerator` class in the static method `trimTypeCode`.

PolicyCenter uses this both for generating the typelist and later when mapping values during policy retrieval.

It is preferable to entirely avoid having codes that exceed the maximum length because there is no way to guarantee that a trimmed code will be unique. If cases occur where trimmed codes are not unique, customize the logic of `trimTypeCode` to use some different approach, or add special cases to generate unique values.

Policy Search SOAP API

There is another web service called `PolicySearchAPI`. It contains only one method. This web service is WS-I compliant. Its one method finds and returns a policy period's public ID by a policy number and a date on which to search.

Call the `findPolicyPeriodPublicIdByPolicyNumberAndDate` method using code like the following:

```
policySearchAPI.findPolicyPeriodPublicIdByPolicyNumberAndDate("123435", Calendar.getInstance())
```

This method performs the search only against policy periods stored in PolicyCenter.

For more general search for policies, use the web service `CCPolicySearchIntegration` mentioned earlier in this topic.

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.

Note: 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:

- Open ClaimCenter to directly view and edit claim information. The file `ClaimDetailsCSV.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, such as:

```
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, such as:

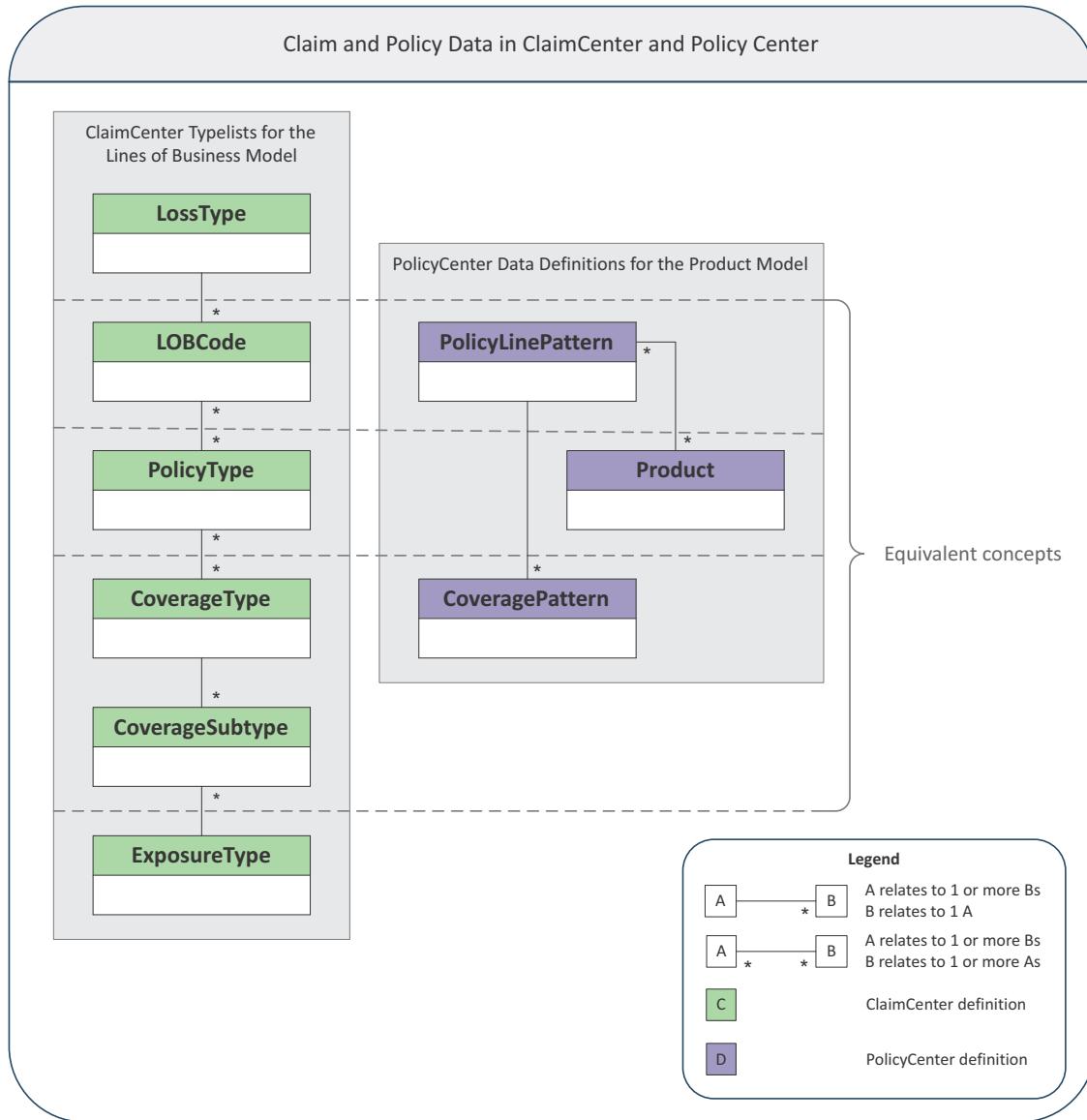
```
http://localhost:8580/bc
```

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 lines of business model synchronized with your product model. When you change your PolicyCenter product model, you must merge the changes with your ClaimCenter lines of business model to keep them synchronized. PolicyCenter provides the ClaimCenter Typelist Generator to help you synchronize your ClaimCenter line of business model with your PolicyCenter product model. The ClaimCenter Typelist Generator is a command line tool.

The ClaimCenter lines of business model uses some 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.

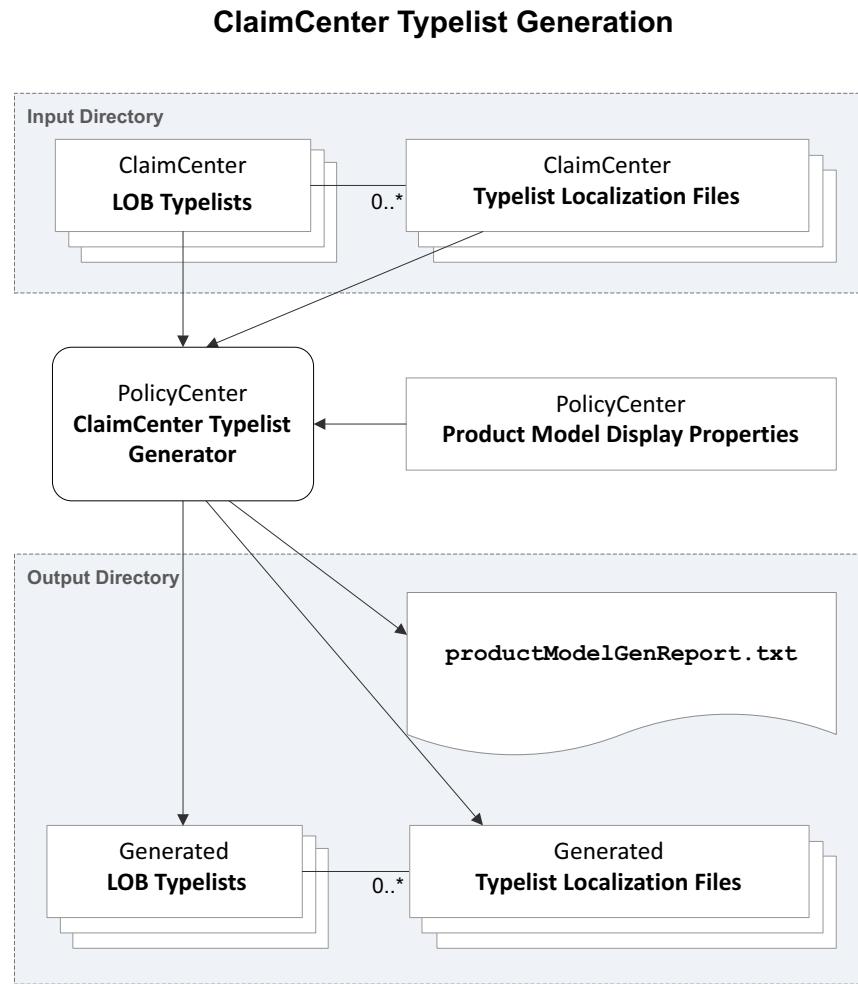
However, the ClaimCenter lines of business model has typelists in its hierarchy that fall above and below its PolicyCenter equivalents. For example, LOB codes in ClaimCenter link to loss types. The generator does not know about loss types. You must link new LOB codes to their parent loss types manually in ClaimCenter Studio. In a similar way at the bottom hierarchy, you must link new coverage types/coverage subtypes from PolicyCenter to exposure types in ClaimCenter.

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. Select this option if you work in development mode or want to demonstrate the products together. For more information, see “Linking PolicyCenter Coverages to the ClaimCenter General Damage Exposure Type” on page 516.

The following diagram illustrates the basic operation of the ClaimCenter Typelist Generator.



Input Files

The ClaimCenter Typelist Generator reads the following ClaimCenter typelist files as input:

- `LOBCode.ttx`
- `PolicyType.ttx`

- CoverageType.ttx
- CoverageSubtype.ttx
- ExposureType.ttx
- CovTermPattern.ttx
- LossPartyType.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 ClaimCenter Studio. 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.

Note: 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.

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 diff tool to determine what typecodes the generator changed.

Development or Demonstration Environment

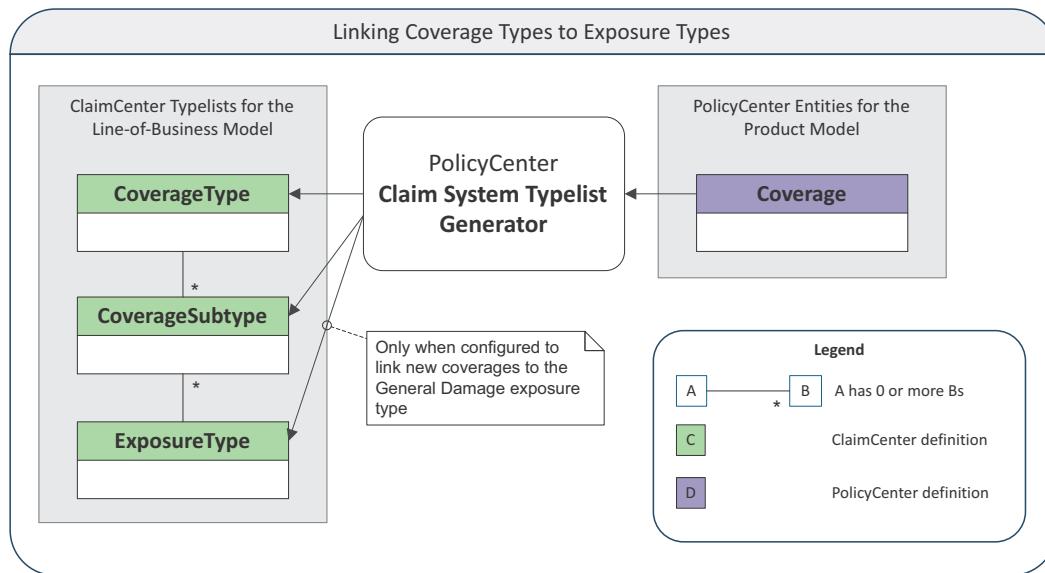
In a development or demonstration environment, you generally set up a PolicyCenter and a ClaimCenter instance on the same machine. If so, the generator can use the ClaimCenter directory that holds the lines of business typelist files as the input and output directories. This avoids steps to manually copy files to and from ClaimCenter. However, this approach does not work if your ClaimCenter and PolicyCenter instances are configured with multiple locales.

Production Environment

In a production environment, do not configure the ClaimCenter Typelist Generator to use ClaimCenter directories for input or output.

Generated Coverage Subtypes

In the ClaimCenter lines of business model, you link coverage types to exposure types through coverage subtypes.



Coverage subtypes essentially duplicate their coverage types. From a technical perspective, they help 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 them as categories of each other. Use the generated subtypes to link corresponding coverage types to exposure types in ClaimCenter Studio.

Linking PolicyCenter Coverages to the ClaimCenter General Damage Exposure Type

You can configure the ClaimCenter Typelist Generator to associate new coverages from PolicyCenter with the General Damage exposure type in ClaimCenter. If you select to map new coverages to general damage exposure when you run the generator, then you must include the typelist file **ExposureType.ttx** in the input directory. The generator updates the **GeneralDamage** typecode by adding new coverage subtypes as categories. You select to map new coverages to general damage exposure by running the generator with **-Dmap_coverages=true**.

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 ClaimCenter Studio. 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 ClaimCenter Studio. 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 knows which codes in ClaimCenter typelists come from PolicyCenter by their source system categories. 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. For more information, see “Typelist Localization” on page 522.

Running the ClaimCenter Typelist Generator

Before you run the ClaimCenter Typelist Generator, you must:

- Know the location of the input and output directories.
- Place the ClaimCenter typelist and typelist localization files in the input directory.

The generator will preserve the lines of business codes that you configured in ClaimCenter Studio or imported from other third-party policy administration systems.

To run the ClaimCenter Typelist Generator

1. At a command prompt, navigate to the *PolicyCenter\bin* directory.

2. Type the following command:

```
gwpc cc-typelist-gen -Dinput_dir=input_dir -Doutput_dir=output_dir -Dmap_coverages=true_or_false
```

Where the command line arguments are:

Argument	Value
-Dinput_dir	Specify the input directory. Required.
-Doutput_dir	Specify the output directory. Required.
-Dmap_coverages	Specify true to map new coverages to general damage exposure. Optional. Default is false.

For more information about the command line arguments, see “Configuring the ClaimCenter Typelist Generator” on page 514.

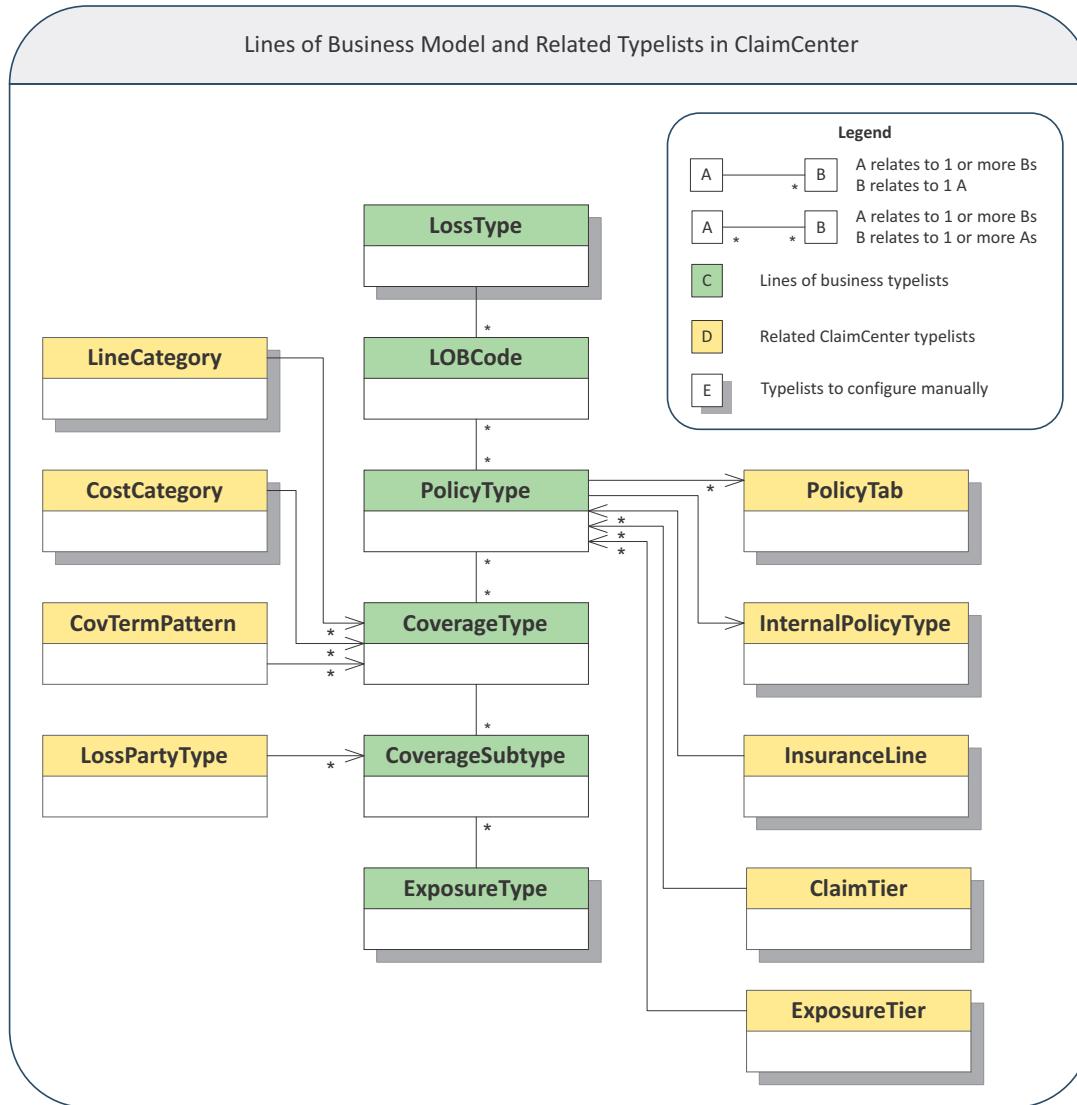
3. Check the output directory for generated files and the generation report *productModelGenReport.txt*.

Use the generation report to:

- Determine success or failure of the generation command
- If the command succeeded, identify new coverage types to map to exposure types in ClaimCenter Studio

Using Generated Typelists in ClaimCenter

After you run the ClaimCenter Typelist Generator, you must perform additional steps to fully merge changes from PolicyCenter with your ClaimCenter lines of business model. The following diagram highlights the typelists that you may need to configure in ClaimCenter Studio.



Suggested steps

1. Make sure to copy generated files to ClaimCenter directories.
2. Use the generation report `productModelGenReport.txt` to determine what coverage types and LOB codes that the generator added.
3. Use a difference-detection (`diff`) tool on your input and output typelist files to determine other typecodes that the generator changed, such as policy types.
4. In ClaimCenter Studio, link generated lines of business typecodes to the top and bottom of the lines of business model:
 - a. For new LOBCode typecodes, add appropriate `LossType` typecodes as parents.
 - b. For new `CoverageSubtype` typecodes, add appropriate `ExposureType` typecodes as children.

For more information, see “Linking New Coverage Types to Exposure Types” on page 520.

5. In ClaimCenter Studio, link generated lines of business codes to related ClaimCenter typelists:
 - a. For new PolicyType typecodes, add PolicyTab typecodes as categories.
 - b. For new PolicyType typecodes, add one InternalPolicyType typecode, commercial or personal, as a category.
 - c. For new PolicyType typecodes, add them as categories to appropriate InsuranceLine, ClaimTier, and ExposureTier typecodes.
 - d. For new CoverageType typecodes, add them as categories to appropriate LineCategory and CostCategory typecodes.

Note that the generator writes an updated CovTermPattern.ttx file, so you do not need to change CovTermPattern in ClaimCenter Studio.

6. If you add or remove CoverageSubtype typecodes, run the generator again with your latest ClaimCenter typelist files. The generator adjusts LossPartyType for you.
7. Add references to new codes in ClaimCenter Gosu classes and other configuration files that relate to coverages types and policy types.
8. 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.

[Copying Generated Files to ClaimCenter](#)

After you run the ClaimCenter Typelist Generator, copy the generated files to ClaimCenter. Skip this step if you configured the generator to use the ClaimCenter directory for lines of business typelist files as its input and output directory.

To copy generated files to ClaimCenter

1. Copy generated typelist files (.ttx) to ClaimCenter/modules/configuration/config/extensions.
2. Copy generated typelist localization files (.properties) to corresponding locale directories in ClaimCenter/modules/configuration/config/locale.

For more information, see “Typelist Localization” on page 522.

[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:

- Coverage subtypes that the generator added to CoverageType that it did not link to exposure types in ExposureType.
- LOB codes that the generator added to LOBType that it 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: 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.
...
```

Values in square brackets are typecodes that were not linked. In ClaimCenter Studio, you must link reported LOB codes to loss types, and you must link reported coverage subtypes to exposure types.

If you choose to link new coverages to the General Damage exposure type when running the generator, the generation report does not include lines for new coverage types. The generator linked them all to typecode GeneralDamage in ExposureType. In this case, use a diff tool and compare input and output versions of CoverageSubtype.ttx to identify new coverage types that you must link to exposure types.

Linking New Coverage Types to Exposure Types

The ClaimCenter Typelist Generator adds generic typecodes in CoverageSubtype.ttx for new coverage types. The generic coverage subtypes have the same codes, names, and descriptions as the corresponding coverages.

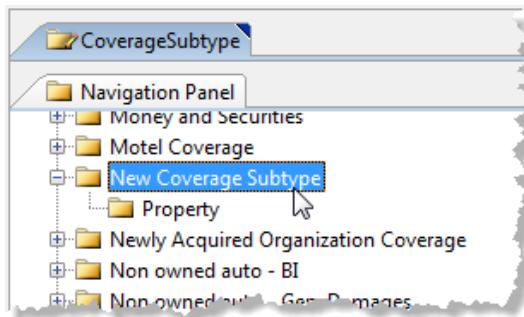
Generally, you want to link a coverage type to a single exposure type. Sometimes, you want to link a coverage to several exposure types. For example, you want liability coverage on a claim to allow exposures for injured people and damaged property.

To link a new coverage to a single exposure type

1. In Project window in ClaimCenter Studio, navigate to Configuration → config → extensions → typelist, and then open CoverageSubtype.ttx.
2. In the list, select the new coverage subtype in the LOB column.
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 click Remove.
4. On the Children folder, click Add.
5. In the Code column, press CTRL+SPACE and select an exposure type from the list.

Result

The new coverage is linked to the exposure type through its generic coverage subtype. In the following example, New Coverage Subtype links to the ClaimCenter Property exposure type.



To link a new coverage to several exposure types

1. In the Resources pane, under the Lines of Business node, select CoverageType.
2. In the Navigation Panel for CoverageType, find the new coverage type and click the plus sign (+) next to it. The panel displays the generic coverage subtype underneath the new coverage type.
3. Right-click the generic coverage subtype, and then click Delete Typecode.
4. For each ClaimCenter exposure type you want to link to the new coverage type:
 - a. In the Navigation Panel for CoverageType, select the new coverage type.
 - b. On the Children (CoverageSubtype) tab, click Add New....

c. In the New CoverageSubtype dialog, specify the following:

Field	Action
Code	Enter a lowercase code that combines the code for the new coverage type with the code for the exposure type you want to link. Abbreviate the two parts of the code if needed. For example, enter nwcvrge-pptydmg for a coverage subtype that links NewCoverageType to PropertyDamage.
Name	Enter a name that combines the name of the new coverage type with the name of the exposure type you want to link. For example, enter New Coverage Type - Property.
Description	Generally, enter the same value that you enter for Name.
Priority	Enter -1.

d. Click OK.

The Children (CoverageSubtype) tab shows the coverage subtype you added.

e. In the Navigation Panel for CoverageType, select your new coverage subtype.

You may need to click the plus sign (+) next to the new coverage type to see your new coverage subtype.

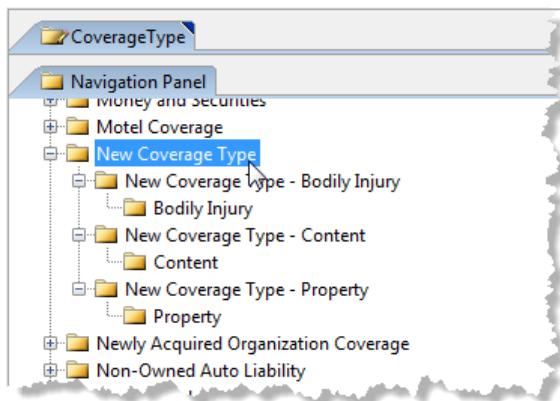
f. In the Children (Exposure) tab, click Add Existing.

g. Press CTRL+SPACE, and then select an exposure type from the list.

h. If you want to link another exposure type to the new coverage type, repeat this procedure beginning at step a.

Result

The new coverage is linked to several exposure types through new coverage subtypes that you added. In the following example, New Coverage Type links to the Bodily Injury, Content, and Property exposure types.



Adding References to New Codes in Gosu Classes and Other 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.

Notes on what to change for 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:
 - `Classes.gw.fnolmapper.acord.impl.AcordExposureMapper.gs`
 - `OtherResources.datatypes.acord.fnolmapper.typecodemapping.xml`
- To support ISO Claim Search for new coverage types, add entries to:
 - `OtherResources.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:
 - `Classes.libraries.PolicyUI.gsx`

Notes on what to change for 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:
 - `Classes.gw.entity.GWClaimTierEnhancement.gsx`
 - `Classes.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:
 - `Classes.libraries.ClaimUI.setInitialValues()`
- To enforce aggregate limits and policy periods for new policy types, review the settings in the following files:
 - `OtherResources.xsds.aggregateLimitUsed-config.xml`
 - `OtherResources.xsds.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. To learn how to set up locales, see “Working with Regional Formats” on page 85 in the *Globalization Guide*.

When you run the generator, it produces updated ClaimCenter localization files with names and descriptions for new typecodes that come from PolicyCenter. 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. This means that localized strings for typecodes that are removed from PolicyCenter remain in the ClaimCenter typelist localization files. The generator cannot distinguish localization properties that refer to deleted PolicyCenter typecode and properties for typecodes that originate from elsewhere.

To use the ClaimCenter TypeList Generator to produce updated typelist localization files

1. In the input directory, create a locale directory for each locale in your Guidewire instances.

For example, your instances have three locales, US English, Canadian English, and Canadian French. Create the following directories in the input directory:

- `INPUT_DIRECTORY/en_CA/`
- `INPUT_DIRECTORY/en_US/`
- `INPUT_DIRECTORY/fr_CA/`

2. Copy files named `typelist.properties` from each ClaimCenter locale directory to corresponding input local directories. ClaimCenter locale directories are located in:

`ClaimCenter/modules/config/locale/`

3. Run the generator.

In the output directory, the generator creates localization folders for each locale and writes an updated `typelist.properties` files in each of them. For example:

- `OUTPUT_DIRECTORY/en_CA/typelist.properties`
- `OUTPUT_DIRECTORY/en_US/typelist.properties`
- `OUTPUT_DIRECTORY/fr_CA/typelist.properties`

4. Copy the typelist localization files from their output directories to the corresponding ClaimCenter locale directories.

IMPORTANT The ClaimCenter Typelist Generator always produces a `typelist.properties` file for the default locale in your PolicyCenter instance, regardless whether you configure the instance for multiple locales. If your ClaimCenter instance has a single locale, then ignore the `typelist.properties` file that the generator produces.

Policy Location Search API

Use the `PolicyLocationSearchAPI` web service to retrieve summary information about policy locations within a rectangular geographic bounding box. This web service is WS-I compliant. The fully qualified path and filename of the default Gosu implementation is:

```
gw.webservice.pc.pc700.policylocation.PolicyLocationSearchAPI.gos
```

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.

See also

- “Installing Sample Data” on page 55 in the *Installation Guide*

Dependencies of the Policy Location Search API

The `PolicyLocationSearchAPI` web service depends on the following:

- The geocoding feature must be enabled.
- The `BatchGeocode` property on policy location `Address` must be set to `true`.

See also

“Using the Geocoding Feature” on page 18 in the *System Administration Guide*.

Finding Policy Locations within Geographic Bounding Boxes

The `PolicyLocationSearchAPI` web service has a single method, with the following syntax:

```
function findPolicyLocationByEffDateAndProductsWithinBoundingBox(effDate : Date,  
    productCodes : String[], topLeftLat : BigDecimal, topLeftLong : BigDecimal,  
    bottomRightLat : BigDecimal, bottomRightLong : BigDecimal) : PolicyLocationInfo[] {
```

The parameters, all required, include:

- 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:
 - Top left latitude
 - Top left longitude
 - Bottom right latitude
 - Bottom right longitude

The bounding box parameters all are `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`.

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.

The base PolicyCenter configuration has built-in support for integration with Guidewire ContactManager. If you have a license for the optional Client Data Management module supported by ContactManager, you can install and integrate ContactManager. For more information, see “Integrating ContactManager with Guidewire Core Applications” on page 49 in the *Contact Management Guide*.

This topic includes:

- “Integrating with a Contact Management System” on page 525
- “Configuring How PolicyCenter Handles Contacts” on page 531
- “Synchronizing Contacts with Accounts” on page 532
- “Account Contact Plugin” on page 533
- “Account Contact Role Plugin” on page 533
- “Contact Web Service APIs” on page 534
- “Address APIs” on page 540

See also

- “Plugin Overview” on page 123
- “Web Services Introduction” on page 37

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` 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.impl.ABContactSystemPlugin`.

IMPORTANT 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. For information on integration with ContactManager, see “Integrating ContactManager with Guidewire Core Applications” on page 49 in the *Contact Management Guide*. For more information about the mapping between PolicyCenter and ContactManager contacts, see “ContactMapper Class” on page 247 in the *Contact Management Guide*.

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:

- 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.

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 ClaimCenter of updates, deletes, and merges of contacts in that external system. When ClaimCenter receives a notification from the external contact management system, ClaimCenter can update its local copies of those contacts to match. See “Contact Web Service APIs” on page 534.

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 default 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.

For more information about messaging and these plugin interfaces, see “Messaging and Events” on page 289

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 the batch server, ClaimCenter 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. For related information, see “Late Binding Data in Your Payload” on page 328.
 - 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.

Contact Retrieval

To support contact retrieval, do the following in your `retrieveContact` method:

1. Determine what object type to populate locally for your contact to connect to the external system.

For example, suppose you define the web service in Studio with the name `MyContactSystem` and that web service has a `MyContact` XML type from a WSDL or XSD file. For this example, suppose your contact retrieval web service returns a `MyContact` object.

Note: For more information about web services and how PolicyCenter imports types from external web services, see “Calling Web Services from Gosu” on page 75.

2. Write your web service code that connects to the external contact management system. Whatever type it returns must contain equivalents of the following `Contact` properties:

`PrimaryPhone`, `TaxID`, `WorkPhone`, `EmailAddress2`, `FaxPhone`,
`HomePhone`, `CellPhone`, `DateOfBirth`, and `Version`.

If the contact is a person, it must have the properties `FirstName` and `LastName`.

If the contact is a company, it must have the property `Name`, containing the company name.

3. Your plugin code synchronously waits for a response.

4. To return the data to PolicyCenter, create a new instance of a contact in the current transaction's bundle.

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.

For example, your code might look like something like this:

```
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  
}  
  
var c = new Contact()
```

5. Populate the new contact entity instance with information from your external contact.

6. Return that object as the result from your `retrieveContact` method.

If you ever need the PolicyCenter implementation to retrieve more fields from the external contact management system, it is best to extend the data model for the `Contact` entity.

Note: If you need to view or edit these additional properties in PolicyCenter, remember to modify the contact-related PCF files to extract and display the new field.

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:

- `ContactIntrinsicType` – The type of contact. To determine if the contact search is for a person or a company, use code such as the following:

```
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.

Note: Properties related to proximity search are unsupported. For example, you cannot search by attitude or longitude using this API.

The second parameter to this method is the `ContactSearchFilter`, which defines some metadata about the search, including:

- 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 `ABContactAPISearchResult` Gosu class in ContactManager. This class is in the package `gw.webservice.ab.ab800.abcontactapi`.

Support for Finding Duplicates

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 Duplicates

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 `ABContactAPIFindDuplicatesResult` class in the package `gw.webservice.ab.ab800.abcontactapi`.

To find duplicates

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 ClaimCenter, typically to store external identifiers.

Alternative Method Signatures 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 ClaimCenter.

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. Just like the `addContact` method, one parameter to `updateContact` is 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.

Overwrite the Local Contact with Latest Values

The previously-mentioned `updateContact` method 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.

It takes two parameters:

- The `Contact` object to update
- The address book UID of the contact in the external system

A typical contact system plugin would perform 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.

Note: This plugin only configures contacts within PolicyCenter. To customize how PolicyCenter interacts with an external system, see “Integrating with a Contact Management System” on page 525.

Configure 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.

Configure 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.

This 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`.

Configure 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.

Configure 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.

Configure 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.

Configure 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.

Note: 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.

IMPORTANT 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:

```
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. This web service is WS-I compliant. Refer to the implementation class in 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 must be 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.pc800.gxmodel.contactmodel.types.complex.Contact) : String {
```

As you look at the fully-qualified path, notice the part of the path before “types”. That indicates the path of the XML model in Studio. In this example, the path is:

PolicyCenter/modules/gsrc/gw/webservice/pc/pc800/gxmodel/ContactModel.gx

View and edit that model as desired to update the XML definition.

See also

- “Web Services Introduction” on page 37
- “The Guidewire XML (GX) Modeler” on page 306 in the *Gosu Reference Guide*

Delete 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.

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.

If you try to delete a contact that is in use, these 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.

Delete a Contact, Specifying Contact by 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 this API:

```
contactAPI.removeContactByPublicID("pc:1234");
```

This method finds all account contacts (*AccountContact* objects) in the database and tries to delete the contact.

Delete a Contact, Specifying Contact by 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 returns nothing.

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 this API:

```
contactAPI.removeContact("12345");
```

Determine if 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 it 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

The following Java example demonstrates this API:

```
Boolean isDeletableContact = contactAPI.isContactDeletableByPublicID("pc:12345");
```

Add 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 objects that represents the contact. The type's fully qualified name is

`gw.webservice.pc.pc800.gxmodel.contactmodel.types.complex.Contact`. This is an XML type defined using the Guidewire XML modeler (GX). See “The Guidewire XML (GX) Modeler” on page 306 in the *Gosu Reference Guide*.

The method returns the public ID of the new contact.

This method updates the contact only if auto sync is enabled for that contact.

The following Java example demonstrates these APIs:

```
gw.webservice.pc.pc700.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");
```

Update Contacts

You can update a contact from an external system by using the `updateContact` method. It returns nothing.

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:

```
gw.webservice.contactapi.beanmodel.XmlBackedInstance
```

Unlike some other XSD-based types used in `ContactAPI`, this type does not contain a corresponding Guidewire XML (GX) model file in PolicyCenter.

The `XmlBackedInstance` is a type defined in the `BeanModel.xsd` file. You can find this XSD file in Studio. The `XmlBackedInstance` type in this XSD is basically a 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.

This method updates the contact only if automatic synchronization (`autosync`) is enabled for that contact.

The following Java example demonstrates these 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 this 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 results are:

- 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 these APIs:

```
contactAPI.mergeContactAddressesByPublicID("pc:1234", "pc:5550", "pc:5551");  
contactAPI.mergeContactAddressesByABUID("12345", "55550", "55551");
```

Merge 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 this 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 these APIs:

```
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.

Note: 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:

- `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 Contacts

To activate or deactivate a contact on all accounts, call the `activateContactOnAllAccounts` method.

Deactivating a contact prevents PolicyCenter from adding that contact to policies. If a now-deactivated contact is already on a policy, it must be removed the next time a job starts on the policy. In other words, this marks contacts as used historically a policy or account but unused for future policies.

You would typically only use the deactivation feature in integrations if you implement the concept of active and inactive contacts in your own external contact management system.

It 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 this API:

```
Integer totalUpdated;  
totalUpdated = contactAPI.activateContactOnAllAccounts("pc:1234", true /* activate */);
```

Get Associated Work Orders for a Contact

You can get associated work orders 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 this 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 work order (job) objects. The job objects are XSD-defined XML objects with the type fully qualified name:

```
gw.webservice.pc.pc700.gxmodel.jobmodel.types.complex.Job
```

This is an XML type defined using the Guidewire XML modeler (GX).

Pass the following parameters in this 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 these APIs:

```
Job[] workOrderList;  
workOrderList = contactAPI.getAssociatedWorkOrdersByPublicID("pc:1234", "BOUND");
```

See also

- See “The Guidewire XML (GX) Modeler” on page 306 in the *Gosu Reference Guide*.

Get Associated Policies for a Contact

Get associated policies for a contact with the `getAssociatedPolicies` method. There are two variants of this 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.pc700.policyperiodmodel.types.complex.PolicyPeriod
```

This is an XML type defined using the Guidewire XML modeler (GX).

The only parameter is the ID of the contact.

The following Java example demonstrates these APIs:

```
gw.webservice.pc.pc700.gxmodel.policyperiodmodel.types.complex.PolicyPeriod[] policyList;  
  
policyList = contactAPI.getAssociatedPoliciesByPublicID("pc:1234");
```

See also

- See “The Guidewire XML (GX) Modeler” on page 306 in the *Gosu Reference Guide*.

Get Associated Accounts for a Contact

Get associated accounts for a contact with the `getAssociatedAccounts` method. There are two variants of this 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.pc700.gxmodel.accountmodel.types.complex.Account
```

This is an XML type defined using the Guidewire XML modeler (GX).

The only parameter is the ID of the contact.

The following Java example demonstrates these APIs:

```
gw.webservice.pc.pc700.gxmodel.accountmodel.types.complex.Account[] accountList;
```

```
accountList = contactAPI.getAssociatedAccountsByPublicID("pc:1234");
```

See also

- See “The Guidewire XML (GX) Modeler” on page 306 in the *Gosu Reference Guide*.

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
<code>updateAddressOnly</code>	Update the address but not its linked addresses
<code>updateAddressAndLinkedAddresses</code>	Update the address and its linked addresses
<code>updateAddressAndUnlink</code>	Update the address and unlink its linked addresses
<code>updateAddress</code>	In the base configuration, it has the same behavior as the <code>updateAddressAndLinkedAddresses</code> method. This method is designed to represent the default behavior for each carrier. Modify this method's implementation as appropriate. Note that the <code>updateAddress</code> method calls the <code>AddressService</code> class. The <code>AddressService.updateAddress(...)</code> 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 following 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 these 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 VII

Importing Policy Data

Zone Import

PolicyCenter includes zone mapping functionality used by the assign by location feature, the address auto-fill feature, and the regional holidays feature. PolicyCenter supports high-volume bulk import of zone data by using database staging tables. This topic describes how database staging table import works to import zone data.

IMPORTANT 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.

This topic includes:

- “Introduction to Zone Import” on page 543
- “Overview of a Typical Database Staging Table Import” on page 546
- “Database Import Performance and Statistics” on page 550
- “Table Import Tools” on page 550
- “Data Integrity Checks” on page 552
- “Table Import Tips and Troubleshooting” on page 553

Introduction to Zone Import

The database staging table for zone data provides significantly higher performance than importing individual records. Database staging table import further improves performance by using bulk SQL Insert and Select statements that operate on the entire zone table at one time.

This topic includes:

- “Staging Tables” on page 544
- “Zone Data” on page 544
- “Integrity Checks” on page 545
- “Load Error Tables” on page 545
- “Load History Tables” on page 545

- “Load Commands and Loadable Entities” on page 545

Staging Tables

Staging tables are database tables that replicate almost completely the 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 directly into the PolicyCenter operational table `pc_zone` by first loading the data into the `pcst_zone` staging table.

Zone Data

PolicyCenter uses zone data for the following features:

- Assignment by location
- Address auto-fill
- Setting regional holidays

Steps to Import Zone Data

The process to import zone data comprises these main steps:

1. **Get or create your zone data files** – Create zone data files in comma separated value (CSV) format that contain columns for the postal code, state, city, and county of specific geographic zones. Each line in the file must use the following format:

```
postalcode,state,city,county
```

For example,

```
94114,CA,San Francisco,San Francisco
```

For more information, see “Zone Data Files Supplied by Guidewire” on page 544.

2. **Run the zone import command to add zone data to the zone staging table** – Run the command line tool `zone_import` in the directory `PolicyCenter/admin/bin`. You can also use the web service `ZoneImportAPI` to add your zone data to the staging table.

For example with the command line tool, your command might look like this:

```
zone_import -clearstaging -import myzonedata.csv -server http://myserver:8080/pc -user myusername
```

For more information, see “Zone Import Command” on page 187 in the *System Administration Guide*.

3. **Run the table import command to bulk load the zone operational table** – Run the command line tool `table_import` in the directory `PolicyCenter/admin/bin`. You can also use the web service `TableImportAPI` to bulk load the zone operational table.

For more information, see “Table Import Command” on page 184 in the *System Administration Guide*.

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 Studio Project window:

```
configuration → config → geodata
```

Within 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. For example, the provided zone data files are not complete and may 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.

Integrity Checks

Before loading staging table data into operational database tables, PolicyCenter runs many PolicyCenter-specific data integrity checks. These checks find and report problems that would cause import to fail or might put PolicyCenter into an inconsistent state. Integrity checks are a large set of auto-generated database queries (SQL queries) built into the application.

You can check if any integrity checks failed at **Server Tools** → **Info Pages** → **Load Errors**. For more information, see “Load Errors” on page 161 in the *System Administration Guide*.

Load Error Tables

The *load error tables* hold data from failed data integrity checks. Do not directly read or write these tables. Instead, examine them using the **Load Errors** interface. You can view these errors in PolicyCenter at **Server Tools** → **Info Pages** → **Load Errors**. For more information about using this screen, see “Load Errors” on page 161 in the *System Administration Guide*.

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
- Error message
- Data integrity check, also called the query, that failed.

In some cases, PolicyCenter cannot identify or store a single LUWID for the error.

Load History Tables

Load history tables store results for import processes, including rows for each integrity check, each step of the integrity check, and row counts for the expected results. Use these to verify that the table-based import tools loaded the correct amount of data. You can view this information in PolicyCenter at **Server Tools** → **Info Pages** → **Load History**. For more information about using this screen, see “Load History” on page 160 in the *System Administration Guide*.

Load Commands and Loadable Entities

PolicyCenter creates staging tables during the upgrade process when the server starts. If a PolicyCenter table is loadable and has at least one loadable property, PolicyCenter creates a corresponding staging table for it.

During staging table import, all loadable entities are copied from the staging tables to the operational tables. After an entity imports successfully, the application sets each entity’s load command ID property (`LoadCommandID`) to correspond to the staging table conversion run that brought the row into PolicyCenter.

An entity's LoadCommandID property is always null for rows that were created in some other way, in other words new entities that did not enter PolicyCenter through staging table import.

The LoadCommandID property persists even after performing additional import jobs. The presence of the LoadCommandID property does not guarantee that the current data is unchanged since the row was imported. If the user, application logic, or integration APIs change the data, the LoadCommandID property stays the same as when the row was first imported.

You can use this feature to test whether an entity was loaded using database staging tables or some other way. From your business rules or from a Java plugin, test an entity's LoadCommandID. From Gosu, check `entity.LoadCommandID`. From Java, check the `entity.getLoadCommandID` method. If the load command is non-null, the entity was imported through the staging table import system. All entities with that same load command ID loaded together in one import request. If the load command is null, the entity was created in some way other than database table import.

These load command IDs correspond to results of programmatic load requests to import staging tables.

For example, use the command line `table_import` tool in the `PolicyCenter/admin/bin` directory. The tool returns the LoadCommandID.

The result of that method is 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. Save that value and test specific entities to see how they were loaded. Compare that value against that saved LoadCommandID. Similarly, if you used the command line tools to trigger database table import, those tools return the LoadCommandID.

You can also track load import history using the database load history tool user interface at `Server Tools → Info Pages → Load History`. For more information about using this screen, see “Load History” on page 160 in the *System Administration Guide*.

Remember that even if an object has a load command ID that matches a known load request, its data may have changed after loading due to user actions or APIs.

Overview of a Typical Database Staging Table Import

To import data by using database staging tables and the database import tools, the server must be set 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 run level.

The database import tools do not require the server to run in a specific mode. You can perform database staging table imports regardless of production, test, or development mode.

Plan when to perform a database import carefully 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 mitigate the amount of down time for users, consider importing large amounts of data in phases rather than in a single operation.

Importing Zone Data

In PolicyCenter, the database staging tables and the database import tools load zone data only. The sources of zone data are CSV files for each country or region that you want to support. Guidewire provides the `zone_import` tool to load zone data from CSV files into the staging table for zone data. As a prerequisite to importing zone data, you must modify `zone-config.xml` files to configure which fields of a zone to data import.

See also

- “Zone Import Command” on page 187 in the *System Administration Guide*

High-Level Steps in a Typical Database Staging Table Import

The following procedure lists the high-level steps involved in a typical database import procedure to load zone data from CSV files into PolicyCenter.

1. Run the `zone_import` command or invoke the web service `IZoneImportAPI` to load zone data from CSV files into the staging table for zone data.
2. If the server mode is production, set the server to the maintenance run level.
3. Run integrity checks on staging table data using the `table_import` command or invoke the `TableImportAPI` web service.
4. Set the server to the multiuser run level.
5. View load errors on the **Load Errors** page.
6. Fix errors and run integrity checks again, repeating some or all of the steps listed previously in this list.
7. After you are certain all integrity checks succeeded, set the server to the maintenance run level.
8. Load the data from the staging tables into operational tables by using web services or command line table import tools.
9. Set the server to the multiuser run level.

Detailed Steps in a Typical Database Staging Table Import

Migrating data from a source or legacy system to PolicyCenter typically happens in the following steps:

1. **If you changed the data model, run the server to perform upgrades** – The staging table import tools require that legacy data be converted to the same format as the server 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 line tool or the web service `SystemToolsAPI.setRunLevel` method.
3. **Back up the operational tables in the database** – Back up operational tables before import.

WARNING As with any major database operation, back up all operational tables before importing.

4. **Clear the staging tables, the error tables, and the exclusion tables** – Your conversion tool would typically do this manually, but you can also do this using web service APIs described in “Table Import Tools” on page 550.
5. **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 API `SystemToolsAPI` method `checkDatabaseConsistency`, or using the command line:
`system_tools -password password -checkdbconsistency`
See “System Tools Command” on page 180 in the *System Administration Guide* for more information about the command line tool.
If you do not run consistency checks regularly, run these a long time before converting your data. For example, plan several weeks to correct any errors you may encounter.
6. **Create zone data files** – Create zone data files in comma separated value (CSV) format. For more information about zone data files, see “Zone Data” on page 544.
7. **Run the zone import tool** – Run the command line tool `zone_import` or use the `IZoneImportAPI` web service to import your CSV data. For more information, see “Zone Import Command” on page 187 in the *System Administration Guide*.

- 8. Request integrity checks from the table import tools** – These tools are available as a web service or from the command line. An integrity check ensures the data meets PolicyCenter basic data integrity requirements. If problems are found, those records are saved in an error table. PolicyCenter runs all integrity checks and reports all errors before stopping.

The command line tool is as follows:

```
table_import -integritycheck
```

Your conversion tool might automatically trigger the integrity check using the web service or command line tools after converting and loading the data into staging tables. At the time you request an integrity check, you can optionally clear error tables and exclusion tables using optional parameters. Also, you can choose to perform the integrity check as part of a load request, and if so the load proceeds only if no integrity check errors occur.

There is an optional command line flag to allow references to existing non-admin rows in the database: -
`allreferencesallowed` (corresponding to the web service boolean parameter
`allowRefsToExistingNonAdminRows`). Only use this flag (or for the web service, set it to `true`) if absolutely necessary. For example, in the rare case a policy period overlaps between the existing operational data and the data you are loading. This option flag can cause performance degradation during the check and load process.

- 9. Set the server to the multiuser run level** – Set each PolicyCenter server to the multiuser run level by using the `system_tools` command line tool or the web service `SystemToolsAPI.setRunLevel` method.

- 10. Check load errors from the Load History page** – This page is available at `Server Tools`→ `Info Pages` → `Load History`. For more information about using this page, see “Load History” on page 160 in the *System Administration Guide*.

- 11. Set the server to the maintenance run level** – Set each PolicyCenter server to the maintenance run level by using the `system_tools` command line tool or the web service `SystemToolsAPI.setRunLevel` method.

- 12. Repeat integrity checks until they succeed** – Repeat integrity checks until there are no errors.

- 13. Load staging tables into operational tables** – Eventually, integrity checks succeed for all records except for records in the exclusion table. At this point, you can load data from the staging tables into the operational tables used by PolicyCenter. Do this using one of the various web service (SOAP) APIs or command line tools for loading. See “Table Import Tools” on page 550. For example, use the table import web service method `integrityCheckStagingTableContentsAndLoadSourceTables`. As PolicyCenter inserts rows into the operational tables, it also inserts results into the load history table.

There is an optional command line flag to allow references to existing non-admin rows in the database: -
`allreferencesallowed` (corresponding to the web service boolean parameter
`allowRefsToExistingNonAdminRows`). Only use this flag (or for the web service, set it to `true`) if absolutely necessary. For example, in the rare case a policy period overlaps between the existing operational data and the data you are loading. This option flag can cause performance degradation during the check and load process.

If you use Oracle databases, Guidewire recommends you use the Boolean parameter `updateDBStatisticsWithEstimate` set to `true`. This indicates to update database statistics on any table loading with estimated changes based on contents of the associated staging table. This corresponds to the optional `-estimateorastats` option for the command line tool. If you load large amounts of data, the source table grows significantly. The optimizer could choose a bad query plan based on the existing state of the database statistics. Avoid this situation by updating the database statistics to reflect the expected size of the table after the load completes.

In other words, for non-Oracle databases, load staging tables with the command:

```
table_import -integritycheckandload
```

For Oracle databases, load staging tables with the command:

```
table_import -estimateorastats -integritycheckandload
```

WARNING For Oracle databases, failing to add the `-estimateorastats` option (or the equivalent web service parameter set to `true`) extremely reduces database performance. Remember to always add this additional option for Oracle databases.

The server automatically removes all data from staging tables at completion. If there were errors, data remains in the staging tables.

14. **PolicyCenter populates user and time properties, as well as other internal or calculated values** – During import, PolicyCenter sets the value of the `CreateUserID` and `UpdateUserID` properties to the user ID of the user that authenticated the web service or command line tool. To detect converted records in queries, create a PolicyCenter user called `Conversion` (or something like that) to detect these records. Creating a separate user for conversion helps diagnose potential problems later on. This user must have the SOAP Administration permission to execute the web service or command line tool. Do not give this user additional privileges or access to the user interface or other portions of PolicyCenter. At this step, PolicyCenter sets the `CreateTime` and `UpdateTime` properties to the start time of the server transaction. All rows now have the same time stamp for a single import run.
15. **Set the server to the multiuser run level** – Set each PolicyCenter server to the multiuser run level by using the `system_tools` command line tool or the web service `SystemToolsAPI.setRunLevel` method.
16. **Review the load history** – Review the load history at `Server Tools → Info Pages → Load Errors`. Ensure that the amount of data loaded is correct. For more information, see “Load Errors” on page 161 in the *System Administration Guide*.
17. **Update database statistics** – Particularly after a large conversion, update database statistics. See “Configuring Database Statistics” on page 40 in the *System Administration Guide*.
18. **Update database consistency checks again** – Assuming there were no consistency errors before importing, new consistency errors after imports indicate something was wrong in the staging table data uncaught by integrity checks. See step 5 for the data consistency check options.

In general, never interact with the PolicyCenter database directly. 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. However, the staging tables are exceptional because conversion tools that you write can read/write staging table data before import.

WARNING 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 “Using the Server Tools” on page 145 in the *System Administration Guide*.

PolicyCenter attempts to perform the integrity check and then load all data in the staging tables, so each import attempt must start with a clean set of tables. Therefore remove the staging data after successfully loading the data.

As part of doing a conversion from a source system into PolicyCenter, there are several ways in which errors can be handled. During the development of the conversion tool, errors frequently occur due to incorrect mapping data from the source system into the staging tables. Errors also occur if you do not properly populating all necessary properties. Typically, fix these errors by adjusting algorithms in your conversion tool. Typically, you run many trial conversions and integrity checks with iterative algorithm changes before finally handling all issues.

If the server flags an error that is simply a problem with the source data, then correct it directly in the staging tables using direct SQL commands. In contrast, never modify operational tables with direct SQL commands.

Alternatively, you may want to correct errors in the source system. In that case, remove records that fail the integrity check from the staging tables. Correct the errors in the source system. Then, rerun the entire conversion process.

Database Import Performance and Statistics

To isolate your code performance from PolicyCenter database import performance, take the following steps:

- If using Oracle, take statistics snapshots (statspack snapshots at level 10 or AWR snapshots) at the beginning and end of `integritycheck` commands and `integritycheckandload` commands.
- Generate the statistics reports (statspack reports or AWR reports) and debug any performance problems. Guidewire recommends you download the Oracle AWR/Oracle Statspack from **Server Tools** → **Info Pages** along with **Load History Info** to debug check and load performance problems.
- Save a copy of the **DatabaseCatalogStatistics** page and archive it before each database import attempt.

IMPORTANT Designing appropriate statistics collection into all database import code dramatically improves the ability for Guidewire to advise you on performance issues related to database import.

PolicyCenter has an API to update database statistics for the staging tables. Use the `TableImportAPI` web service method `updateStatisticsOnStagingTables` or use the command line tool command:

```
table_import -updatedatabasestatistics
```

Table Import Tools

PolicyCenter provides a set of staging table import functions to help you with the data import process. PolicyCenter exposes the staging table import tools in the following ways:

- **Table import web service** – You can use table import methods defined in the `TableImportAPI` web service. All tools are provided as synchronous methods, which do not return until the command completes. Some tools have an additional asynchronous method, which returns immediately and then performs the command as a batch process. The asynchronous versions have method names that end in “`AsBatchProcess`”. For example, `deleteExcludedRowsFromStagingTablesAsBatchProcess`.

For general information about logging into and using web services, see “Web Services Introduction” on page 37.

- **Table import command line tools** – You can use table import options of the `table_import` command line tool. For a list of options, run the following command from `PolicyCenter/admin/bin` and view the built-in help.

```
table_import -help
```

For more information about the `table_import` command, see “Table Import Command” on page 184 in the *System Administration Guide*.

All servers in your PolicyCenter cluster must be at the maintenance run level to call any of these import functions. The maintenance run level ensures that no end users are on the system. This prevents new data added through the user interface from interfering with bulk data imports from the staging tables.

Use the following command for each server in the cluster to set this run level.

```
system_tools -maintenance -server url -password password
```

IMPORTANT All table import commands require all PolicyCenter servers in a cluster to be at the maintenance run level or the table import command fails.

Alternatively, set the run level by using the `SystemToolsAPI` web service method `setRunLevel`.

The following sections briefly describe the most useful table-based import tools.

Integrity Check Tool

This tool performs the integrity check only. It does not attempt to load the operational tables. Optionally, the tool clears the error table before starting and load the exclusion table with the distinct list of logical unit of work IDs in the error table. This is the default behavior.

This simplifies the integrity check and increases the performance of the operation. If you cannot accept this limitation, then set the `allowRefsToExistingNonAdminRows` option.

- API method: `integrityCheckStagingTableContents`
- Command line option: `integritycheck`

Integrity Check And Load Tool

This tool runs the integrity check process and, if no errors are detected, proceeds with loading the operational tables. In addition, if you are using Oracle only, you can instruct the system to update database statistics on each table after data is inserted into it. This helps the database avoid bad queries caused by large changes in the size of tables which occur during import.

- API method: `integrityCheckStagingTableContentsAndLoadSourceTables`
- API method: `integrityCheckStagingTableContentsAndLoadSourceTablesAsBatchProcess`
- Command line option: `-integritycheckandload`

This tool updates estimated row and block counts on the table and indexes. This helps avoid potential optimizer issues if you reference tables with a size that qualitatively changed in the same transaction.

The Boolean parameter `updateDBStatisticsWithEstimate` updates database statistics on any table loading with estimated changes based on contents of the associated staging table. This corresponds to the optional `-estimateorastats` option for the command line tool. If you load large amounts of data, the source table grows significantly. The optimizer could choose a bad query plan based on the existing state of the database statistics. Avoid this situation by updating the database statistics to reflect the expected size of the table after the load completes. Use this feature only with Oracle databases. Otherwise, PolicyCenter ignores this parameter.

Clear Error Table Tool

Deletes all rows from the error table, typically in preparation for a new integrity check.

- API method: `clearErrorTable`
- Command line option: `-clearerror`

Clear Staging Table Tool

Deletes all rows from the staging table, typically in preparation for a new integrity check.

- API method: `clearStagingTables`
- Command line option: `-clearstaging`

Update Statistics

This tool updates the database statistics on all staging tables.

- API method: `updateStatisticsOnStagingTables`
- Command line option: `-estimateorastats`

Other Import-Related Tools

The following tools are available for system-wide settings during table import. For more information about the command line tools, see the “PolicyCenter Administrative Tools” on page 173 in the *System Administration Guide*. For more information about logging into and using web service APIs, see “Web Services Introduction” on page 37

Setting Run Level

Set the run level to maintenance before performing a table import.

- API method: `SystemToolsAPI.setRunLevel(SystemRunlevel.GW_MAINTENANCE);`
- Command line option: `system_tools -password password -maintenance`

Checking Operational Table Consistency

Check the operational database consistency before running a table import.

- API method: `SystemToolsAPI.checkDatabaseConsistency(returnAllResults);`
- Command line option: `system_tools -password password -checkdbconsistency`

IMPORTANT If you do not run consistency checks regularly, run them long before starting conversion. Allow several weeks to correct any errors you may encounter.

Even between corresponding tables there are differences in the columns. For example, compare the `pc_policy` and `pcst_policy` tables. There are a number of differences:

For example, click on an entity in the *Data Dictionary* and look at each property to see whether the dictionary says “(loadable)”. If it does, that property is included in the staging table for that entity.

.Foreign key fields must contain the public ID (`publicID`) of the target entity.

Data Integrity Checks

Before loading staging table data into the operational database tables, PolicyCenter runs a broad set of PolicyCenter-specific data integrity checks. These checks find and report problems that would cause the import to fail or put PolicyCenter into an inconsistent state.

PolicyCenter requires that data integrity checks succeed as the first step in the load process. This means that even if errors are found and these rows were removed, PolicyCenter still requires rerunning integrity checks before your data is reloaded.

Contrast data integrity checks with other validations:

- Integrity checks check different things from requirements that the user interface (PCF) code enforces. For example, a property that is nullable in the database may be a property that users must set in the PolicyCenter user interface. Importing a `null` value in this property is acceptable for database integrity checks. However, if you edit the object containing the property in the PolicyCenter interface, PolicyCenter requires you to provide a non-`null` value before saving because of data model validation.
- Integrity checks are different from validation rule sets and the validation plugin.

Examples of Integrity Checks

The following list is a partial list of data integrity checks that PolicyCenter enforces during database import:

- No duplicate `PublicID` strings within the staging tables or in the corresponding operational tables
- No unmatched foreign keys
- No missing, required foreign keys
- No invalid codes for type key properties
- No invalid subtypes
- No `null` values in non-`null` (operational) properties that do not provide a default. Empty strings and text containing only space characters are treated as `null` values in data integrity checks for non-nullable properties.
- No duplicate values for any unique indexes.

For a full list of integrity checks, see the [Load Integrity Checks](#) page within the PolicyCenter Server Tools tab. You can view all integrity check SQL queries. For more details, see “Load Integrity Checks” on page 160 in the *System Administration Guide*.

Why Integrity Checks Always Run Before Loading

There are many reasons for PolicyCenter to rerun integrity checks during any load request, even in situations in which the conversion tool believes that it fixed all load errors. For example:

- If errors were found during population of the operational tables, the entire process must roll back the database. Rolling back the database changes typically is slow and resource-intensive. It is much better to identify problems initially rather than trigger exceptions during the load process that require rolling back changes.
- Even if you remove all error rows, integrity check violations can occur for certain errors that cannot be tied to a single row. Because some errors cannot be tied to a particular row, there is no associated logical unit of work ID.

Table Import Tips and Troubleshooting

Some things to know about importing using the staging tables and safely and successfully running the process:

- All staging table import commands require the servers to be in maintenance mode, formally known as the maintenance run level. This prevents users from logging into the PolicyCenter application.
- PolicyCenter runs the load process inside a single database transaction to be able to roll back if errors occur. This means that a large rollback space may be required. 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 load. After a successful load, PolicyCenter provides table-specific update database statistics commands in the `cc_loaddirstatisticscommand` table. You can selectively update statistics only on 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. Assuming there were no consistency errors before importing, consistency errors after an import indicates that there is something wrong with the data in the staging tables uncaught by integrity checks. See “[Overview of a Typical Database Staging Table Import](#)” on page 546 for example commands.
- During PolicyCenter upgrade, the PolicyCenter upgrader tool may drop and recreate the staging tables. This occurs for any staging table in which the corresponding operational table changed and requires upgrade.

WARNING Never rely on data in the staging tables remaining across an upgrade. Never perform a database upgrade during your import process.

Other Integration Topics

Archiving Integration

PolicyCenter supports archiving policy terms as serialized streams of data, with one serialized stream per `PolicyPeriod` object in the term. You can store the serialized data in a file, in a document storage system, or in a database as binary large object. Each serialized stream is an XML document.

This topic includes:

- “Overview of Archiving Integration” on page 557
- “Archiving Storage Integration Detailed Flow” on page 562
- “Archive Retrieval Integration Detailed Flow” on page 565
- “Archive Source Plugin Utility Methods” on page 566
- “Check for Archiving Before Accessing Policy Data” on page 568
- “Upgrading the Data Model of Retrieved Data” on page 568

See also

- “More Information on Archiving” on page 329 in the *Application Guide* for a list of topics related to archiving.

IMPORTANT Guidewire strongly recommends that you contact Customer Support before implementing archiving.

Overview of Archiving Integration

PolicyCenter supports archiving policy terms. The application generates XML documents for the data. 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 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` interface, which the `PolicyPeriod` entity type implements. Some archiving APIs use the `RootInfo` interface 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` page to reset the `PolicyPeriod.ExcludedFromArchive` flag to false.

There are two archiving-related plugin interfaces in PolicyCenter. Both implementations are in the `gw.plugin.archive.impl` package. The following table summarizes the role of each interface:

Plugin interface	Purpose	Demonstration implementation class
<code>gw.plugin.archiving.IArchiveSource</code>	Storing and retrieving a serialized XML document that represents one <code>PolicyPeriod</code> graph. For more information, see “Archive Source Plugin” on page 560.	<code>PCArchivingRecordingPlugin</code>
<code>gw.plugin.archive.IPCArchivingPlugin</code>	Determines when to archive a policy term. See “Archiving Eligibility Plugin” on page 561	<code>PCArchivingPlugin</code>

Archiving Integration and Archiving Eligibility Flow

The high-level integration flow for archival storage:

1. The PolicyCenter archiving batch process runs.
2. PolicyCenter determines the set of `PolicyPeriod` objects that need archiving. To determine what `PolicyPeriod` objects to skip, PolicyCenter uses the following properties:
 - `PolicyTerm.NextArchiveCheckDate` - a date that determines the earliest archive eligibility. PolicyCenter populates this property from calling the policy eligibility plugin. “Archiving Eligibility Plugin” on page 561.

- `PolicyPeriod.ExcludedFromArchive` - a Boolean flag that if set to `true` prevents ClaimCenter from even attempting to archive the claim. If the value is `true`, ClaimCenter skips the claim and does not run the archiving rule set for this claim again until this flag is reset to `false`. For an excluded claim, the `ClaimInfo.ExcludedReason` property contains a `String` that describes the reason for the exclusion.
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 `PolicyPeriod` graph into an in-memory XML representation.
 5. PolicyCenter serializes each XML document.
 6. PolicyCenter calls a customer-written archive source plugin to store the data.
 7. The archive source plugin sends archive data to an external system. You implement this plugin to connect to your own archive source.
 8. 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, see “Overview of Archiving Integration” on page 557 for details.

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` objects 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 archiving storage plugin to retrieve the XML format archive data from the external archive.
4. PolicyCenter de-serializes the XML data into an in-memory representation of the stored XML.
5. If the archive data is from an earlier data model of PolicyCenter, PolicyCenter next runs upgrade steps on the object. This includes 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 568
6. PolicyCenter converts the temporary object into a real entity, upgrades the data model if necessary, and finally persists it to the database.
7. PolicyCenter adds activities to notify retrieval requestors of the success. There may be more than one requestor to retrieve one policy term.

The following diagram summarizes the elements of this system.



Archive Source Plugin

Customers using archiving must implement the `IArchiveSource` plugin interface. The responsibility of this plugin is storage and retrieval of archive data from the backing store. The backing store typically is on a different physical computer. The backing store could be anything.

Common choices for a backing store include:

- Files on a remote file system
- A document management system document
- A database row containing a large binary object

PolicyCenter includes a demonstration implementation of this interface:

`gw.plugin.archive.impl.PCArchivingRecordingPlugin`. This implementation writes archive data as files to the server's local file system.

The superclass of the built-in plugin implementation is the base class `gw.plugin.archiving.ArchiveSource`.

IMPORTANT Guidewire provides an archive source plugin implementation for demonstration purposes only. Do not use this in a production system. Implement your own archiving plugin that stores data on an external system.

To store archive data, PolicyCenter calls the plugin's `store` method synchronously, waiting for it to complete or fail.

To retrieve archive data, PolicyCenter calls the plugin's `retrieve` method. The only argument the `retrieve` method gets is a root info object, which is an instance of an entity type that implements the `RootInfo` interface. A root info object encapsulates a summary of one archived object. The root info object remains in the database after archiving most of the data.

In PolicyCenter, the root info object always is a `PolicyPeriod` object. In the APIs or the documentation where you see the interface type `RootInfo`, this refers to `PolicyPeriod`. Your plugin implementation must store enough reference information into the `PolicyPeriod` object and its associated `EffectiveDatedFields` object to determine how to retrieve any archived document from its backing store.

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` outside the main database transaction for the storage request. If the storage request fails, all PolicyCenter data changes never commit to the database. However, any changes you make during the call to `storeFinally` do persist to the database. For details, see “Archive Source Plugin Storage Methods” on page 563.

WARNING Be extremely careful that you understand potential failure conditions and the relationship between each archive method and associated database transactions.

For details for each plugin method, refer to reference information grouped by purpose of the methods:

- “Archive Source Plugin Storage Methods” on page 563
- “Archive Source Plugin Retrieve Methods” on page 566
- “Archive Source Plugin Utility Methods” on page 566

Archiving Eligibility Plugin

The `IPCArchivingPlugin` plugin interface determines only when to archive a policy term. There are additional built-in algorithms for determining eligibility of archiving. For example, if the term has open jobs, no policy periods will be archived on that term. However, the `IPCArchivingPlugin` plugin implementation gives you an opportunity to set the archiving date even later than the built-in logic.

There is a built-in demonstration implementation called `PCArchivingPlugin`.

There is only one method in this plugin interface. Implement the `getArchiveDate` method, which takes a `PolicyTerm` object. This method must calculate the earliest date that PolicyCenter can archive this policy term. The method must return the date as a `Date` object, trimmed to represent midnight.

The demonstration implementation checks for open claims and returns the maximum date of the set of the following:

- all the claim dates for open claims
- today

For more implementation details of the default behavior, see the following:

- the demonstration implementation class `PCArchivingPlugin`
- “Selecting Policy Terms for Archive Eligibility” on page 468 in the *Configuration Guide*

Archiving Web Services in PolicyCenter

To manipulate archiving from external systems, call the `IArchiveAPI` web service. This web service is WS-I compliant. For example, check if a policy term is archived, or suspend a policy from further archiving. For more information, see “Archiving Web Services” on page 119.

Set Encryption Plugin for Encrypted Properties in Archived Objects

The `config.xml` configuration parameter `DefaultXmlExportEncryptionId` specifies the unique encryption ID of an encryption plugin.

If archiving is enabled, the application uses that encryption plugin to encrypt any encrypted fields during XML serialization. For more about encryption unique IDs, see “Writing Your Encryption Plugin” on page 241.

WARNING To avoid accidentally archiving encrypted properties as unencrypted data, be sure to set the parameter `DefaultXmlExportIEncryptionId`.

Archiving Storage Integration Detailed Flow

PolicyCenter implements archive integration as a work queue.

Archive Writers and Workers

PolicyCenter performs the following archiving tasks to create archive writers and workers:

1. First, the application confirms that the network archive source system is available. It is possible that the archive source is unavailable due to network problems or configuration issues. The archive source plugin returns its status in its `getStatus` method. If the archive source 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 source is available, the batch process called Archiving Item Writer finds objects that are potentially eligible for archiving. The batch process creates a work item as a row in the database for every object that is eligible for archiving.

In PolicyCenter, from the work queue perspective, the eligible object for archiving is a policy term. However, a term contains potentially multiple `PolicyPeriod` objects 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. Some eligibility criteria is defined in internal code.

Before attempting archiving, PolicyCenter re-checks some criteria to see if it is eligible even if it already checked this when 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 all 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 object that implements the `RootInfo` interface. 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 objects within the domain graph of this object. To accomplish this task, the worker *tags* the archived root object (the `PolicyPeriod`) specified in the work item. Next, worker recursively tags all entities in the domain graph whose parent object was tagged.

The process of tagging includes setting the `ArchivePartition` property on the root object 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 subobjects.

- 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` 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, see “Overview of Archiving Integration” on page 557 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` object.
5. The worker commits the database transaction. This ends the database transaction for the preceding database changes. If any changes before this step threw an exception, all changes in this entire transaction are rolled back.
 6. As the last step in the archive 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 “Running Code in an Entirely New Bundle” on page 347 in the *Gosu Reference Guide*.

See also

- “Scheduling Work Queue Writers and Batch Processes” on page 117 in the *System Administration Guide*
- “Scheduling Work Queue Writers and Batch Processes” on page 117 in the *System Administration Guide*

Error Handling During Archive

If the archive process fails in any way, consult both of the following:

- Application logs
- The `Archive Info` page within the `Server Tools` page.

To view the `Archive Info` page, you must set the `config.xml` configuration parameter `ArchiveEnabled` to `true`.

Archive Source Plugin Storage Methods

PolicyCenter calls various `IArchiveSource` methods during the archive store process. The following list of methods defines the main storage methods:

- `prepareForArchive(info : RootInfo)`
- `updateInfoOnStore(info : RootInfo)`
- `store(graph : InputStream, info : RootInfo)`
- `storeFinally(info : RootInfo, finalStatus : ArchiveFinalStatus, cause : List<String>)`

For PolicyCenter, when you see the interface type `RootInfo` in documentation or code in Studio, this always refers to a `PolicyPeriod` entity instance.

`prepareForArchive(info : RootInfo)`

The method call for `prepareForArchive` occurs outside the archive transaction. Thus:

- PolicyCenter does not roll back any changes if the archiving operation fails.
- PolicyCenter does not commit any 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 the domain graph actually archives successfully. The method has no transaction of its own. If you want to update data, then you must create a bundle and commit that bundle yourself.

In the demonstration plugin implementation, this method does nothing.

updateInfoOnStore(info : RootInfo)

PolicyCenter calls the `updateInfoOnStore` method after the application preps 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 info object. See the following topic on the `store` method below.

PolicyCenter calls this method inside the archiving transaction. This enables you to make additional updates on `RootInfo` objects.

As the method call is inside the archiving transaction, if that transaction fails, then PolicyCenter does not commit any updates made during the call to the database.

Depending on what kind of action you need to do in `updateInfoOnStore`, you might need similar or complementary changes in the plugin methods `updateInfoOnRetrieve` and `updateInfoonDelete`.

In the demonstration plugin implementation, this method does nothing.

WARNING You can modify the root info object (`PolicyPeriod`) object in the `updateInfoOnStore` method. However, it is dangerous and unsupported to modify any other objects in the policy period graph in this method.

store(graph : InputStream, info : RootInfo)

The `store` method is the main configuration point for taking XML data (in the form of an `InputStream`) and transferring it to an external archive source system.

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 archive 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 external archive backing store.

The archive process passes in the `RootInfo` object 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 root info object. Normally, it is best to make those changes in the `updateInfoOnStore` method.

Generally speaking, it is best to change no entity data at all in the `store` method. Within the `store` method (or any other part of the data base transaction), the only properties that are safe to change are the non-array properties on the `RootInfo` object. 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` object. Only make this change from the `store` method if your external archive source 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 objects that you set up in your `updateInfoOnStore` method call.

storeFinally(info : RootInfo, finalStatus : ArchiveFinalStatus, cause : List<String>)

PolicyCenter calls this method outside the archiving transaction after completing that transaction. The `finalStatus` parameter value indicates if the archiving delete operation was successful. Check this value. This allows the archive storage system to reverse any changes that were not part of the transaction in the rare case in which the delete transaction fails.

To change any data, you must perform any entity instance modification within a call to `Transaction.RunWithNewBundle(block)`. See “Running Code in an Entirely New Bundle” on page 347 in the *Gosu Reference Guide*.

If this method is called with errors, `finalStatus` is something other than `success`. The `cause` parameter contains a list of `String` objects that describe the cause of any failures. The `String` objects are the text for the exception cause hierarchy.

This permits two different design strategies:

- The recommended technique is to send and commit your data in the external source in the `store` method. On failure conditions, in `storeFinally` you can back out of any changes in the external data store. For example, delete any temporarily-created files.
- Alternatively, you can perform a two stage commit. In other words, send the data to the external system in the `store` method but finalize it in `storeFinally` if and only if `finalStatus` indicates success.

It is important to be careful about what kinds of work you do in the `storeFinally` method to properly handle error conditions. If the `storeFinally` method throws an exception, the application logs the exception but the main database transaction already completed. 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.

Archive Retrieval Integration Detailed Flow

As one part of the retrieve process, *retrieval* is the step of getting data from an archive source back into the main database. It is up to you provide the necessary hooks into the retrieve process.

Within PolicyCenter, archive document retrieval has the following overall flow:

1. The user identifies an object to retrieve.
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` object. 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.
4. PolicyCenter creates any data that relates to the retrieve. For example, notes.
5. PolicyCenter performs any upgrade steps defined for the data. See “Upgrading the Data Model of Retrieved Data” on page 568.
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, this method sets the `PolicyTerm.NextArchiveCheckDate` property to a date based on the `ArchiveDaysRetrievedBeforeArchive` configuration parameter. This ensures that the application does not immediately archive the object again but instead waits until it is eligible again.
7. PolicyCenter commits the bundle.
8. PolicyCenter calls `IArchiveSource.retrieveFinally`. 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 source. Guidewire does **not** recommend, nor does it support, the use of this method to make changes to data after you retrieve it. 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, for example, to retrieve objects that do not pass validation rules. Committing additional changes in `retrieveFinally` could fail validation, causing only those changes to be rolled back, not the entire retrieve request.

WARNING Do not change the retrieved data in `retrieveFinally`.

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](#) page. If you cannot view the [Archive Info](#) page, you must set the configuration parameter `ArchiveEnabled` to `true` in `config.xml`.

See also

- “Archiving in PolicyCenter” on page 323 in the *Application Guide*
- “Archiving Web Services” on page 119

Archive Source Plugin Retrieve Methods

PolicyCenter calls the following `IArchiveSource` methods during the archive retrieval lifecycle:

- `retrieve(info : RootInfo) : InputStream`
- `updateInfoOnRetrieve(info : RootInfo)`
- `retrieveFinally(info : RootInfo, finalStatus : ArchiveFinalStatus, cause : List<String>)`

For PolicyCenter, when you see the interface type `RootInfo` in documentation or code in Studio, this always refers to a `PolicyPeriod` entity instance.

`retrieve(info : RootInfo) : InputStream`

The `retrieve` method is the main configuration point for retrieving XML data from the external archive source. You must return the data as an `InputStream`. The archive process passes the `RootInfo` object 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 backing store.

`updateInfoOnRetrieve(info : RootInfo)`

The retrieval process calls this method within the retrieval transaction. This occurs after the archive process populates the bundle of the `RootInfo` object with the objects produced by parsing the XML returned by the earlier call to `retrieve(RootInfo)`.

From within the `updateInfoOnRetrieve` method, you can modify the new objects in the object graph as well as the `RootInfo` object.

`retrieveFinally(info : RootInfo, finalStatus : ArchiveFinalStatus, cause : List<String>)`

The retrieval process calls this as the last retrieval step, outside of the retrieve transaction. The `finalStatus` parameter tells if the retrieve was successful. Use this plugin method to perform other actions on the storage. For example, deleting the file, or marking its status.

Archive Source Plugin Utility Methods

PolicyCenter calls these utility methods as needed during the archive process:

- `updateInfoonDelete(info : RootInfo) : List<Pair<IEntityType, List<Key>>`

- `delete(info : RootInfo)`
- `retrieveSchema(platformMajor : int, platformMinor : int, appMajor : int, appMinor : int, extension : int) : InputStream`
- `storeSchema(platformMajor : int, platformMinor : int, appMajor : int, appMinor : int, extension : int, schema : InputStream)`

For PolicyCenter, when you see the interface type `RootInfo` in documentation or code in Studio, this always refers to a `PolicyPeriod` entity instance.

Your plugin must implement all of these methods. Refer to the demonstration plugin for guidance.

`updateInfoonDelete(info : RootInfo) : List<Pair<IEntityType, List<Key>>>`

At some later time after archiving, a user or batch process may decide to delete the archived object entirely, including the data that remained in the local database. This is called *purgung*. During purging, you may want to delete the archived data on the external store, but you are not required to do so.

During purging, PolicyCenter calls the `updateInfoonDelete` method to determine additional objects that PolicyCenter must also delete if it deletes the object 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` object and related entities from the active database.

If your archive source plugin creates extension entity objects in the `updateInfoOnArchive` method that link to the `RootInfo` object, you must return these objects from `updateInfoonDelete`. This causes the application to delete those extension entity instances.

WARNING Declaring these relationships in `updateInfoonDelete` is important because you may have created other objects that PolicyCenter cannot discover by looking at the foreign key references on the `RootInfo` object.

The return type is a list of pair (`Pair`) objects. Each pair object is parameterized on:

- An entity type
- A list of Key objects. The Key contains the relevant `object.ID` value for that object.

Typically, it is inappropriate to make entity changes in this method. If this method does make changes, the application commits those changes before calling the plugin method called `delete`. If you create objects in the `updateInfoOnArchive` that link to the `RootInfo` entity, return those objects in the return results of this method.

Generally speaking, if you mark an object to delete by returning it from this method, you also return all objects that link to those objects.

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 object A links to object B, which links to the `RootInfo` object, you must return A before B.

Typically you would determine the correct order of types and then delete all objects of that object type all at once.

However, although usually unnecessary, it is supported to return one entity type more than once in the return list. Theoretically in complex configuration edge cases, this might be necessary to enforce proper ordering of the deletion process. For example, this allows you to enforce the following deletion order:

1. Delete object ID 100 of type Type1.
2. Delete object ID 900 of type Type2.
3. Delete object ID 101 of type Type1, a type already mentioned.

delete(info : RootInfo)

In the `delete` method, your plugin implementation must delete from the backing store the data identified by the specified `RootInfo` object. 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 object.

The delete process calls this method *after* PolicyCenter deletes associated objects return from `updateInfoonDelete`. See earlier in this topic for details of `updateInfoonDelete`.

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`. The XSD describes the format of the XML files for that version.

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.

Check for Archiving Before Accessing Policy Data

You must be careful before accessing policy data to determine whether the data is archived. Before traversing a `PolicyPeriod` to its subobjects, you must check whether that policy period is archived. If a policy period is archived, objects are no longer accessible other than the `PolicyPeriod` and its associated effective dated fields object. If your code attempts to traverse the objects 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. For details, see “Gosu Code and Archiving” on page 473 in the *Configuration Guide*.

Upgrading the Data Model of Retrieved Data

After the archiving plugin retrieves archived 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 entity, represented by the interface `IArchivedEntity`. The application runs upgrade trigger steps on the object. Upgrade steps such as adding a column happen on this archived entity object.

Only after all upgrade triggers run, PolicyCenter attempts to convert the data to a real entity that matches the current data model of the application. If this process fails, the entire restore from archive fails. If it succeeds, the application commits the new data to the main application database.

See also

- For more about writing upgrade triggers, see “Customizing the Upgrade” on page 160 in the *Upgrade Guide*.

Custom Batch Processing

PolicyCenter lets you implement custom batch processing to supplement the batch processing provided in the base configuration.

This topic includes:

- “Overview of Custom Batch Processing” on page 569
- “Developing Custom Work Queues” on page 572
- “Example Work Queues” on page 579
- “Developing Custom Batch Processes” on page 582
- “Example Batch Processes” on page 586
- “Enabling Custom Batch Processing to Run” on page 589
- “Monitoring Batch Processing” on page 591
- “Periodic Purging of Batch Processing Entities” on page 593

See also

- “Batch Processing” on page 111 in the *System Administration Guide*

Overview of Custom Batch Processing

Custom batch processing that you implement in PolicyCenter operates like the batch processing that PolicyCenter provides in the base configuration.

This topic includes:

- “Styles of Batch Processing” on page 570
- “Choosing a Style for Custom Batch Processing” on page 570
- “Nightly and Daytime Batch Processing” on page 570
- “Batch Processing Typecodes” on page 571

Styles of Batch Processing

PolicyCenter supports two styles of batch processing:

- **Work queue** – A work queue operates on a batch of items in parallel. Work queues run partially on the active batch server and can run on other servers in a PolicyCenter clustered environment.

A work queue comprises the following components:

- A processing thread, known as a *writer*, that selects a batch of items to process
- A queue of selected work items
- Processing threads, known as *workers*, that process the items to completion

Work queues are suitable for high volume batch processing that requires items to be processed in parallel to achieve an acceptable throughput rate.

- **Batch process** – A batch process operates on a batch of items sequentially. Batch processes run only on the active batch server in a PolicyCenter clustered environment.

Batch processes are suitable for low volume batch processing that achieves an acceptable throughput rate when it processes items sequentially. 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 Style for Custom Batch Processing

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 on separate execution threads distributed across multiple servers in a PolicyCenter clustered environment.

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.

Nightly and Daytime Batch Processing

Specific types of batch processing fall into one of two broad categories:

- **Nightly batch processing** – Processes business transactions accumulated at the end of specific business periods, such as business days, months, quarters, or years
- **Daytime batch processing** – Defers to periodic asynchronous background processing complex transactional processing triggered by user actions

Depending on the category of your custom batch processing, certain implementation details can vary.

Nightly Batch Processing

Nightly batch processing typically processes relatively large batches of work, such as processing premium payments that accumulate during the business day. Each type of nightly batch processing runs once during the night, when users are not active in the application.

Because nightly batch processing typically operates on large batches and has rigid processing windows, develop your nightly batch processing as a custom work queue. Nightly batch processing often requires processing items in parallel to achieve sufficient throughput, and the workloads of nightly batch processing are typically too severe for the batch server.

For nightly 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 nightly batch processing typically has many worker threads 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.

Nightly batch processing frequently requires chaining so that completion of one type of batch processing starts another type of follow-on batch processing. Regardless of implementation style – work queue or batch process – you most likely must develop custom process completion logic for your type of nightly batch processing.

See also

- “Performing Custom Actions After Batch Processing Completion” on page 122 in the *System Administration Guide*

Daytime Batch Processing

Daytime batch processing typically processes relatively small batches of work, such as reassigning activities escalated by users. Daytime processing runs frequently during the day, while users are active in the application. You might schedule different types of daytime batch processing 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 batch processing often achieves sufficient throughput by processing items sequentially, its workload on the batch server can slow overall performance of the application. As a work queue, your daytime batch processing can be assigned to a worker thread on a server other than the batch server.

If you develop your daytime batch processing as a custom work queue, you typically can use the standard work queue table for your work items. Different types of daytime batch processing run intermittently and in different bursts of relatively short work. So, table contention between various types of daytime batch processing is often minimal.

Daytime batch processing seldom requires chaining. Completion of one type of daytime batch processing seldom starts another type of follow-on batch processing. Regardless of implementation style, you most likely do not need to develop custom process completion logic for your type of daytime batch processing.

Batch Processing Typecodes

PolicyCenter identifies and manages 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 let you 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 line or web service
- A case clause in the Batch Processing Completion plugin for your type of batch processing

Regardless the style of batch processing you implement, first define a new typecode in the `BatchProcessType` typelist for your type of batch processing.

See also

- “Defining the Typecode for Your Custom Work Queue” on page 574

Developing 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 in a batch in parallel. Develop a custom work queue for batch processing that requires processing the units of work in a batch in parallel to achieve an acceptable throughput rate.

This topic includes:

- “Custom Work Queue Overview” on page 572
- “Defining the Typecode for Your Custom Work Queue” on page 574
- “Defining the Work Item Type for Your Custom Work Queue” on page 574
- “Creating Your Custom Work Queue Class” on page 575
- “Developing the Writer for Your Custom Work Queue” on page 576
- “Developing the Workers for Your Custom Work Queue” on page 577

See also

- “Example Work Queues” on page 579

Custom Work Queue Overview

A custom work queues comprises the following components:

- **Writer** – The `findTargets` Gosu method on a class that extends `WorkQueueBase` to select 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` Gosu method on the same class that extends `WorkQueueBase` base class to process 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.

Your Custom Work Queue Class

You implement the code for your writer and your workers as methods on a single Gosu class. You must derive you 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 must override a 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 items 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 where 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.

Workers of a Work Queue

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.

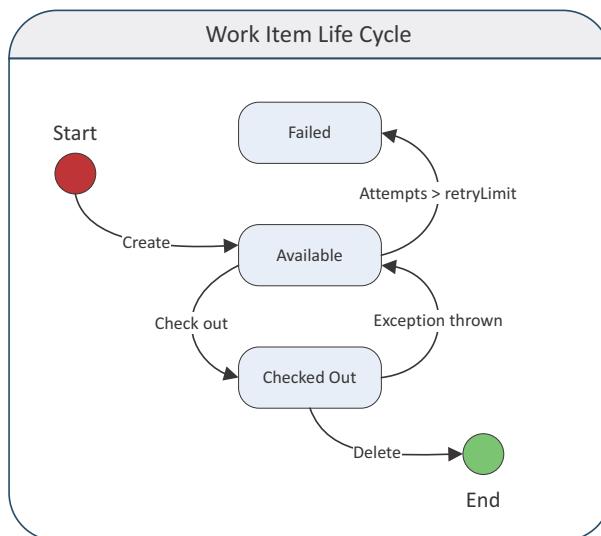
Work Queues and Work Item Entity Types

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, so workers can access them asynchronously, in parallel, and distributed on different servers.

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.

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 life-cycle.



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 when two 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. As long as failed work items remain in a work queue, the writer will not select another batch. Someone must take corrective action and remove all failed work items from the work queue before the next batch can run.

See also

- For more information on the lifecycle of work queues, see “Troubleshooting Work Queues” on page 124 in the *System Administration Guide*.

Defining the Typecode for Your Custom Work Queue

Before you begin developing the Gosu code for your custom work queue, define your type of custom work queue. Add a typecode for it in the `BatchProcessType` typelist. The constructor of your custom work queue class requires the typecode as an argument.

1. In the Project window in Studio, navigate to `configuration` → `config` → `Extensions` → `Typelist`, and then open `BatchProcessType.ttx`.
2. Click **Add Typecode** .
3. In the panel of fields on the right, specify the following values.

Field	Description
<code>code</code>	Specify a code that uniquely identifies your type of custom batch processing, for example <code>SendReminderEmail</code> . The code identifies your type of batch processing in configuration files. The code also is a parameter to the maintenance tools command and web service that allows someone to start your type of custom batch processing.
<code>name</code>	Specify a human readable identifier for your type of custom batch processing, for example <code>Send reminder email</code> for overdue activities. This name appears for your writer on the Batch Process Info screen.
<code>desc</code>	Provide a short description of what your custom batch processing accomplishes, for example, <code>Send reminder email for overdue activities</code> . This description appears for your writer on the Batch Process Info screen.

See also

- “Categorizing Your Batch Processing Typecode” on page 590

Defining the Work Item Type for Your Custom Work Queue

Before you develop the Gosu code for your custom work queue, decide whether to use the `StandardWorkItem` entity type for your work queue table or a custom work item type. Define a custom work item type for the following reasons:

- Include custom fields on work items not available on the `StandardWorkItem` entity type
- Avoid table contention with other work queues that process typically 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 custom work queue types for work queues with typically large batches that potentially run at the same as other work queues with typically large batches. Especially if you develop a custom work queue for nightly batch processing, consider developing a custom work queue type instead of using `StandardWorkItem`. If you create a custom work item type 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`.

1. In the **Project** window in Studio, navigate to **configuration** → **config** → **Extensions** → **Entity**.
2. Right-click **Entity**, and then select **New** → **Entity**.
3. Enter a name and description for your work item type, and then click **OK**.
4. In the drop down list next to , select **implementsEntity**.
5. In the **name** field on the right, select **WorkItem** from the drop down list.
6. For each field on your custom work item type, do the following:
 - a. select the field type in the drop down list next to .

For example, you might select **column**.

 - b. In the panel of the fields on the right, specify the values appropriate to the field type.

For example, you might specify the following values for a column that represents an `Activity` instance as the sole unit of work for your custom work queue.

Field	Description
<code>name</code>	Specify <code>Activity</code> .
<code>type</code>	Select <code>softentityreference</code> .
<code>null</code>	Select <code>false</code> .

Whenever you define a custom work item for a work queue, you must implement the `createWorkItem` method to write them to the queue.

See also

- “Writing Custom Work Item Types” on page 577

Creating Your Custom Work Queue Class

After you define a typecode for your custom work queue and possibly create a custom work item type for it, you are ready to create your custom work queue class. This class contains the programming logic for the writer and the workers of your work queue. You must derive your class from `WorkQueueBase`, and you generally must override the following two methods:

- `findTargets` – The logic for the writer, which selects units of work for a batch and returns an iterator for the result set

- `processWorkItem` – The logic for the workers, which operates on a single unit of work selected by the writer

The `WorkQueueBase` provides other methods that you override in certain circumstances, but generally you can 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`.

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 `Activity` as the target, or unit of work, type. It also declares that `StandardWorkItem` as the work queue type.

```
class MyWorkQueue extends WorkQueueBase <Activity, StandardWorkItem> {
```

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 your 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, or 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

You provide the programming logic for a writer thread by overriding the `findTargets` method that your custom work queue class inherits from `WorkQueueBase`.

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.

You 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.PropertyInfo.Name, 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

If you defined a custom work item type instead of using `StandardWorkItem`, you must override the `createWorkItem` method to write new work items to your custom work queue table. The 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, too.

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) : WorkItem {
    var customWorkItem = new myWorkItem(safeBundle)
    customWorkItem.Activity = activity
    return workItem
}
```

Developing the Workers for Your Custom Work Queue

You provide the programming logic for a worker thread by overriding the `processWorkItem` method that your custom work queue class inherits from `WorkQueueBase`. 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 work item is the type and the target type are the types that you specified in the constructor of your custom work queue class derived from `WorkQueueBase`. At the time PolicyCenter calls the `processWorkItem` method, the `Status` field of the work item is set to `checkedout`.

Successful Work Items

When your worker finishes operating on a target instance, return from the `processWorkItem` method. 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,
        null, // To:
        "Activity not viewed for five days", // From:
        "Take a look at activity " + target.Subject + ", due on " + target.TargetDate + ".") // Subject:
        "Take a look at activity " + target.Subject + ", due on " + target.TargetDate + ".") // Body:

    return
}

```

In your override of the `processWorkItem` method, specify the same work item entity type that you specified in the constructor of your custom work queue class.

Updating Entity Data

The `processWorkItem` method does not have a bundle to manage database transactions related to targeted units of work. A bundle exists at the time PolicyCenter calls your `processWorkItem` method, but that bundle is for updates that PolicyCenter makes to the work item. To modify entity data, you must use the `Transaction.RunWithNewBundle` API to create a bundle.

IMPORTANT You must use the `RunWithNewBundle` API to modify entity data with your custom work queue. For more information, see “Running Code in an Entirely New Bundle” on page 347 in the *Gosu Reference Guide*.

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
}

```

Failed Work Items

If your worker code encounters an error while operating on a target instance, throw an exception. Some types of exceptions are thrown automatically, such as a concurrent data change exception (CDCE). They generally resolve themselves quickly and automatically. Other logic errors that your code can detect may 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 `processWorkItem` method. Whenever PolicyCenter catches an exception, it increments the `Attempts` property on the work item. If the value of the `Attempts` property then exceeds the value of `WorkItemRetryLimit` in `config.xml`, PolicyCenter sets the `Status` property to `available`, otherwise it sets to `failed`.

Work Item Retry Interval

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 when two worker threads 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 thread, the exceptional condition often is resolved. The second worker then process the work item to completion successfully.

Failing Work Items Immediately

You can provide an implementation of the `handleException` method inherited from the base class to augment the actions that PolicyCenter takes when code in the `processWorkItem` throws an exception. For example, your worker code might throw a custom exception when it encounters a logic error that cannot resolve itself without human intervention. In your implementation of the `handleException` method, you can check the exception type. If the type is your custom exception, your code sets the `Status` property to `failed` regardless of the number of retry attempts. If the exception is any other type, call the super class.

```
override function handleException (W workItem, java.lang.Throwable exception,
    int consecutiveExceptions): void {
    // Fail immediately for a custom work queue exception
    if (exception typeis myCustomWorkQueueException) {
        workItem.Status = failed
        return true
    }
    // Fail only for other exceptions if attempts exceeds retry count
    else
        return super (workItem, exception, consecutiveExceptions)
}
```

See also

- To configure how PolicyCenter handles work items that trigger exceptions, see “Scheduling Work Queue Writers and Batch Processes” on page 117 in the *System Administration Guide*

Example Work Queues

This topic contains the following example work queues:

- “Simple Example of a Work Queue” on page 579
- “Example Work Queue for Updating Entities” on page 580
- “Example Work Queue with a Custom Work Item Type” on page 581

See also

- “Developing Custom Work Queues” on page 572

Simple Example of a Work Queue

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 has not been 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.

```
uses gw.processes.WorkQueueBase
uses gw.api.database.Query
uses gw.api.database.DBFunction
uses java.util.Iterator
uses gw.api.email.EmailUtil

/**
 * 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 MyWorkQueue extends WorkQueueBase <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 (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.PropertyInfo.Name, 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): void {
    // Extract an object referencet to the unit of work: an Activity instance
    var activity = extractTarget(WorkItem) // Convert the ID of the target to an object reference
    if (activity.AssignedUser.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 " + target.Subject + ", due on " + target.TargetDate + ".") // Body:
    }
    return
}
}

```

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.processes.WorkQueueBase
uses gw.api.database.Query
uses gw.api.database.DBFunction
uses java.util.Iterator
uses gw.transaction.Transaction
uses gw.api.email.EmailUtil

< /**
 * 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 MyWorkQueue extends WorkQueueBase <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 (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.PropertyInfo.Name, 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): void {
        // Extract an object referencnt 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,
                null, // To:
                "Activity not viewed for five days", // From:
                "See activity " + target.Subject + ", due on " + target.TargetDate + ".") // Subject:
                // 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 Gosu code is an example of a custom work queue that uses a custom work item type for its work queue table. So, 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. The work queue uses a custom work item type, so it uses the type its class declaration and constructor.

```

uses gw.processes.WorkQueueBase
uses gw.api.database.Query
uses gw.api.database.DBFunction
uses java.util.Iterator
uses gw.transaction.Transaction
uses gw.api.email.EmailUtil

/**
 * 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 MyWorkQueue extends WorkQueueBase <Activity, MyWorkItem> {

    /**
     * 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 (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, LessThanOrEquals,
        java.util.Date.Today.addBusinessDays(-5))

    return targetsQuery.select().iterator()
}

/**
 * Write a custom work item
 */
override function createWorkItem (activity : Activity, safeBundle : Bundle) : WorkItem {
    var customWorkItem = new MyWorkItem(safeBundle)
    customWorkItem.Activity = activity

    return workItem
}
/**
 * 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,
            null, // To:
            "Activity not viewed for five days", // From:
            "See activity " + target.Subject + ", due on " + target.TargetDate + ".") // Subject:
            "See activity " + target.Subject + ", due on " + target.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 Processes

A *batch process* is code that runs without human intervention as background process on a single server to process the units of work in a batch sequentially. Develop a custom batch process for batch processing that can achieve an acceptable throughput rate by processing the units of work in a batch sequentially on a single thread.

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.

This topic includes:

- “Custom Batch Process Overview” on page 583
- “Creating a Custom Batch Process” on page 583

See also

- “Example Batch Processes” on page 586

Custom Batch Process Overview

Custom work queues are Gosu classes that extend the `BatchProcessBase` base class. The `BatchProcessBase` base class includes code that helps PolicyCenter manage the batch processing of your custom batch process class.

Note: Custom batch processes are not intended as 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 implementations of the following methods:

- `doWork` – Select the units of work for a batch and process them sequentially on the single execution thread that the `DoWork` method provides.
- `incrementOperationsCompleted` – After completing a unit of work, increment the count of completed items for the process history of the batch.
- `incrementOperationsFailed` – If a unit or work failed to process successfully, increment the count of failed items for the process history of the batch.

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. The methods `incrementOperationsCompleted` and `incrementOperationsFailed` are not thread-safe. Entity instances in transactional bundles on separate threads may have problems. If your type of batch processing 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`.

Creating a Custom Batch Process

To create a custom batch process

1. In Studio, edit the `BatchProcessType` typecode.
 - a. Add an element to represent your own batch process.
 - b. In the typecode editor, for the new typecode, add one or more categories as they apply for your batch process. Use the `BatchProcessTypeUsage` typelist with following values:
 - `UIRunnable` - the process is runnable both from the user interface and from the web service APIs.
 - `APIRunnable` - the process is only runnable from web service APIs.
 - `Schedulable` - the process can be scheduled.
 - `MaintenanceOnly` - only run the process if the system is at maintenance run level.You must add at least one category or else your batch process cannot run.
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. Do your batch processes work in this method. For more information about this class, including some optional methods to override, see “Batch Process Implementation Using the Batch Process Base Class” on page 584.
3. In Studio, in the Plugins editor, register your new plugin for the `IProcessesPlugin` plugin interface. For more information, see “Implementing the Processes Plugin” on page 591.

4. If you want your custom batch process to run regularly on a schedule instead of on demand, add the entry to the XML file `scheduler-config.xml`. For more information, see “Scheduling Work Queue Writers and Batch Processes” on page 117 in the *System Administration Guide*.

Batch Process Implementation Using the Batch Process Base Class

Your custom batch process must extend the `BatchProcessBase` Gosu class. You must override the `dowork` method, which takes no arguments. You can override other methods, but 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(...)` API to create a bundle.

IMPORTANT To modify entity data in your batch process, you must use the `RunWithNewBundle` API. For more information about creating bundles, see “Running Code in an Entirely New Bundle” on page 347 in the *Gosu Reference Guide*.

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. For more information, see “Request Termination” on page 584.

IMPORTANT PolicyCenter calls the `requestTermination` method in a different thread from the thread that runs your batch process `dowork` method.

The following subtopics list other property getters or methods that you can call or override.

Check Initial Conditions

PolicyCenter instantiates batch process classes at server startup. Later, PolicyCenter calls the `checkInitialConditions` method to determine whether to start your batch process. If the method returns `true`, PolicyCenter starts your batch process by calling its `dowork` method. If the method returns `false`, initial conditions are not met and PolicyCenter does not call the `dowork` method, skipping the batch process for the time being.

The batch process base class always returns `true` from `checkInitialConditions`. Override this method if you want to provide a conditional response. If you override `checkInitialConditions`, 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 on 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.

Request Termination

If a user clicks the `Stop` button on the Batch Process Info page, this requests the batch process to terminate. PolicyCenter calls the batch process `requestTermination` method to terminate a batch process if possible.

Your 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 batch process base class 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`. When the user requests termination of the batch process, the application calls your overridden version of the method.
- Ensure that your main `doWork` method frequently checks 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 system log to indicate whether the batch process says it could terminate. 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.

Exclusive

The property getter for the `Exclusive` property for a batch process determines whether another instance of this batch process can start while this process is running. The base class implementation always returns `true`. Override this 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 this `false` 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 `false`. For example,

```
override property get Exclusive() : boolean {  
    return false  
}
```

Note: For Java implementations, you must implement this property getter as the method `isExclusive`, which takes no arguments.

Description

The `getDescription` method gets the batch type's description. The base class gets this string from the batch type typecode description. Override this method if you need to customize this behavior.

Detail Status

The property getter for the `DetailStatus` property gets the batch type's detailed status. The base class defines a default simple implementation. Override the default property getter to provide more useful detail information about the status of your batch process for the Administration user interface. The details might be important if your class experiences any error conditions.

Note: For Java implementations, you must implement this property getter as the method `getDetailStatus`, which takes no arguments.

Progress Handling

The `Progress` property in the batch process dynamically returns the progress of the batch process. The base class returns text in the form "x of y" where x is the amount of work completed and y is the total amount of work. If y is unknown, returns just "x". These values are determined from the `OperationsExpected` and `OperationsCompleted` properties. From Java, this is the `getProgress` method.

The following list includes related properties and methods you can use that the base class implements:

- `OperationsExpected` property - a counter for how many operations are expected, as an `int` value. From Java this counter is the `getOperationsExpected` method.
- `OperationsCompleted` property - a counter for how many operations are complete, as an `int` value. From Java this counter is the `getOperationsCompleted` method.
- `incrementOperationsCompleted` method - this no-argument method increments an internal counter for how many operations completed. 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. The user interface can display this number or debugging code can track this number to see the progress. This method returns the current number of operations completed. For each entity for which you have an error condition, call this method once to track the progress of this batch process.
- `OperationsFailed` property - an internal counter for the number of operations that failed. From Java this counter is the `getOperationsFailed` method.
- `incrementOperationsFailed` method - this method 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. The user interface can display this number or debugging code can track this number to see the progress. This method returns the current number of operations failed. You must also call the `incrementOperationsCompleted` method.

IMPORTANT For any operations that fail, call both the `incrementOperationsFailed` method and the `incrementOperationsCompleted` method.

-
- `Finished` property - return a Boolean value to indicate whether the process completed. Completion says nothing about the errors if any. From Java, this is the `isFinished` method.

Type

The base class maintains a `Type` property, which contains the batch process type, as a `BatchProcessType` type-code. From Java this property is the `getType` method.

Example Batch Processes

This topic contains the following example batch processes:

- “Example Batch Process for a Background Task” on page 587

- “Example Batch Process for Unit of Work Processing” on page 587

See also

- “Developing Custom Batch Processes” on page 582

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.

```
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("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. For more information, see “Running Code in an Entirely New Bundle” on page 347 in the *Gosu Reference Guide*.

Example Batch Process for Unit of Work Processing

The following Gosu code is an example of a custom batch process as a type of batch processing that processes units of work rather than operating as background task. Its process history tracks the number of items that processed successfully or that failed. Its units of work are urgent activities.

The following Gosu batch process implements a notification scheme such that any urgent activities must be handled within 30 minutes. If it is not handled within that time range, the batch process notifies a supervisor. If still not resolved in 60 minutes, it sends a message further up the supervisor chain.

If there are no qualified activities, it returns false so that it will not create a process history. If there are items to handle, it increments the count. The application uses this count to display batch process status “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 `TerminateRequested` to terminate the loop if the user or the application requested to terminate the process.

In this Gosu example, it does not actually send the email. Instead it prints to the console. You can change this to use the real email APIs if desired.

```

package sample.processes
uses gw.processes.BatchProcessBase
uses java.util.Map
uses gw.api.util.DateUtil
uses java.lang.StringBuilder
uses java.util.HashMap
uses gw.api.profiler.Profiler

class TestBatch extends BatchProcessBase
{
    static var tag = new gw.api.profiler.ProfilerTag("TestBatchTag1","A sample tag")
    var work : ActivityQuery

    construct() {
        super( "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 {
            work = find(a in Activity where a.Priority == "urgent" and a.Status == "open"
                and a.CreateTime < DateUtil.currentDate().addMinutes( -30 ) )
            OperationsExpected = work.getCount()
            var map = new HashMap<Contact, StringBuilder>()
            for (activity in work) {
                if (TerminateRequested) {
                    return;
                }
                incrementOperationsCompleted()
                var haveContact = false
                var msgFragment = constructFragment(activity)
                haveContact = addFragmentToUser(map, activity.AssignedUser, msgFragment) or haveContact
                var group = activity.AssignedGroup
                haveContact = addFragmentToUser(map, group.Supervisor, msgFragment) or haveContact
                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 {
        return "http://localhost:8080/pc/Activity.go(${activity.id})"

        // TODO: you must ADD A PCF ENTRYPPOINT THAT CORRESPONDS TO THIS URL TO DISPLAY THE ACTIVITY.
    }
}

```

```

private function addFragmentToUser(map : Map<String, 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(email, 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</td><th>Supervisor</th></tr>" +
        body +
        "    </table>" +
        "  </body>" +
        "</html>"
    email.addToRecipient(new EmailContact(contact))
    EmailUtil.sendEmailWithBody(null, email);
}
}

```

To use this entry point, use the following PCF entry point in the file `PolicyCenter/modules/configuration/pcf/EntryPoints/Activity.pcf`:

```

<PCF>
<EntryPoint authenticationRequired="true" id="Activity" location="ActivityForward(actvtIdNum)">
<EntryPointParameter locationParam="actvtIdNum" type="int"/>
</EntryPoint>
</PCF>

```

Also include the following `ActivityForward` PCF file as the `ActivityForward.pcf` in each place in the PCF hierarchy that you would like it:

```

<PCF>
<Forward id="ActivityForward">
<LocationEntryPoint signature="ActivityForward(actvtIdNum : int)" />
<Variable name="actvtIdNum" type="int"/>
<Variable
    initialValue="find(a in Activity where a.ID == new Key(Activity, actvtIdNum))"
    name="actvt"
    type="Activity"/>
<ForwardCondition action="PolicyForward.go(actvt.Policy);"
    ActivityDetailWorksheet.goInWorkspace(actvt)"/>
</Forward>
</PCF>

```

IMPORTANT You must use the `runWithNewBundle` API if want to modify entity data in your custom batch process. For more information, see “Running Code in an Entirely New Bundle” on page 347 in the *Gosu Reference Guide*.

Enabling Custom Batch Processing to Run

Before your type of custom batch processing can run, you must perform the following procedures:

- “Categorizing Your Batch Processing Typecode” on page 590
- “Updating the Work Queue Configuration” on page 590
- “Implementing the Processes Plugin” on page 591

The procedures to perform depend on whether you implemented a custom work queue or a custom batch process.

See also

- To learn how to run your type of custom batch processing on a schedule or on demand, see “Batch Processing” on page 111 in the *System Administration Guide*

Categorizing Your Batch Processing Typecode

For your type of custom batch processing to run, you must 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 category or your type of custom batch processing cannot run.

The runnable categories are:

- `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` - The process can be scheduled.
- `MaintenanceOnly` - Only 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 batch processing type code. Categorizing a batch processing 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

- To learn how to add categories to a typecode, see “Dynamic Filters” on page 298 in the *Configuration Guide*.

Updating the Work Queue Configuration

PolicyCenter instantiates custom work queues that derive from `WorkQueueBase` at server startup. 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="integer" throttleInterval="decimal"
    env="string" server="string"/>
  ...
  <worker instances="integer" throttleInterval="decimal"
    env="string" server="string"/>
</work-queue>
```

The `work-queue` element identifies your custom work queue to PolicyCenter and the batch processing framework. Specify the fully qualified class name of your custom work queue Gosu class for the `workQueueClass` attribute.

Each `worker` element defines the worker threads for a server. To distribute workers among multiple servers, add multiple `worker` elements. You must include one `worker` element with the `instances` attribute set to 1 or greater for your custom work queue to operate.

See also

- For complete information, see “Scheduling Work Queue Writers and Batch Processes” on page 117 in the *System Administration Guide*.

Implementing the Processes Plugin

PolicyCenter instantiates custom batch processes that derive from `BatchProcessBass` at server startup. For PolicyCenter to instantiate your custom batch process, you must implement the Processes plugin (`IProcessesPlugin`). The `IProcessesPlugin` plugin relies on the typecode in `BatchProcessType` typelist for your type of batch processing.

For each typecode in the `BatchProcessType` that represents a custom batch process, PolicyCenter calls the `createBatchProcess` method of the Processes plugin. The method takes a typecode from the `BatchProcessType` as a parameter. Implement the `createBatchProcess` method with a switch statement and a case clause for each type of custom batch process that you develop.

The following example code demonstrates how to implement the `IProcessesPlugin` plugin. The example instantiates two custom batch process.

```
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_CLAIMHEALTHCALC:
                return new ClaimHealthCalculatorBatch();

            case BatchProcessType.TC_PURGEMESSAGEHISTORY:
                return new PurgeMessageHistory(arguments);

            default:
                return null
        }
    }
}
```

Monitoring Batch Processing

Although batch processing runs without human intervention as a background task, people must manage them and monitor their progress.

This topic describes various PolicyCenter features for monitoring default custom batch processing:

- “The Work Queue Info Page” on page 591
- “The Batch Process Info Page” on page 592
- “Monitoring for Batch Processing Completion” on page 592
- “Maintenance Tools” on page 592
- “Process History” on page 592

The Work Queue Info Page

System administrators use the **Work Queue Info** page to control and view information about PolicyCenter and custom work queues.

See also

- “Work Queue Info” on page 147 in the *System Administration Guide*

The Batch Process Info Page

System administrators Use the **Batch Process Info** page to start and view information about PolicyCenter and custom batch processes, including writer processes for work queues.

See also

- “Batch Process Info” on page 146 in the *System Administration Guide*

Monitoring for Batch Processing Completion

Nightly batch processing frequently requires chaining, so completion of one type of batch processing starts another type of follow-on batch processing. Regardless of implementation style – 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 run of batch processing vary depending on the implementation style:

- **Work queues** – Completed if no work items for a batch remain in the work queue, other than work items that failed
- **Batch processes** – Completed if the batch process stopped and its process history is available

PolicyCenter provides Process Completion Monitor batch processing and the `IBatchCompletedNotification` plugin to permit custom code to react to the completion of any type of batch processing.

The `IBatchCompletedNotification` plugin has a `completed` method that you override to perform specific actions if a work queue or batch process completed a batch of work. The parameters of the `completed` method are the `ProcessHistory` and the number of failed items. In the `completed` method, add a case clause for your type custom batch processing by specifying its typecode from the `BatchProcessType` typelist.

See also

- “Performing Custom Actions After Batch Processing Completion” on page 122 in the *System Administration Guide*

Maintenance Tools

You can start, terminate, or get 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
- “Maintenance Tools Command” on page 176 in the *System Administration Guide*

Process History

PolicyCenter creates a process history instance for every run of a work queue or batch process. You can download completed process histories from the `Work Queue Info` and `Batch Process Info` pages and in the administrative user interface.

Batch Processing Style	Download Format
Work Queue	CSV
Batch Process	HTML

Periodic Purging of Batch Processing Entities

Entities related to batch processing, such as process history and work items, accumulate after batch processing runs have been completed. Outdated entities for completed batches must be purged periodically to avoid slowing performance of all types of batch processing.

PolicyCenter provides the following batch processes to purge outdated entities related to batch processing:

- **Process History Purge** – Purges batch process history data from the process history table, the source for download history on the [Batch Process Info](#) page
- **Work Queue Instrumentation Purge** – Purges instrumentation data for work queues, the source for downloadable history on the [Work Queue info](#) page
- **Work Item Set Purge** – Purges work item sets that are more than a specified number of days old
- **Purge Failed Work Items** – Purges failed work items from all work queues

See also

- For more information on the purge processes, see “List of Work Queues and Batch Processes” on page 124 in the [System Administration Guide](#)

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.

This topic includes:

- “Free-text Search Plugins Overview” on page 595
- “Free-text Load and Index Plugin and Message Transport” on page 596
- “Free-text Search Plugin” on page 598

See also

- “Search Overview” on page 345 in the *Configuration Guide*
- “Free-text Search Setup” on page 89 in the *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.

Connecting the Free-text Plugins with 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. Even if you enable the free-text plugins, the implementations do not fully operate unless the `FreeTextSearchEnabled` parameter in `config.xml` is set to `true`. In addition, the `ISolrMessageTransportPlugin` requires you to enable the `PCSolrMessageTransport` message destination. 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`. Whenever you use free-text search in a production environment, set the `debug` parameters for each plugin to `false`.

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 default 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 `PCPCSolrMessageTransportPlugin` 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`.

See also

- “Messaging and Events” on page 289

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 itself is enabled in the Plugins Registry in the default configuration. However, you must enable the relevant messaging destination in the `Messaging` editor in Studio.

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 the following fields are set to the following values.

Field	Value
Enabled	Select the checkbox
Destination ID	69
Name	<code>Java.MessageDestination.SolrMessageTransport.Policy.Name</code>
Transport plugin	<code>ISolrMessageTransportPlugin</code> . This is not the fully-qualified name. This is the name for the plugin in the Plugins Registry editor.
Poll interval	1000

Field	Value
Events	<ul style="list-style-type: none"> • ContactChanged • ContactAdded • ContactRemoved • JobPurged • PolicyAddressChanged • PolicyPeriodAdded • PolicyPeriodChanged • PolicyPeriodPurged • PolicyPeriodRemoved • PolicyPurged • PreemptedPeriod
Chunk Size	• 100000
Poll Interval	• 1000
Max Retries	• 3
Initial Retry Interval	• 1000
Message Without Primary	• Select the Single Threaded radio button

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>true</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`.

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.	true
<code>fetchSize</code>	Determines the maximum number of query results that the Guidewire Solr Extension returns for each search request.	100

See also

- “Modifying Free-text Search for Additional Fields” on page 371 in the *Configuration Guide*

Servlets

You can define simple web servlets for your PolicyCenter application using the `@Servlet` annotation.

Implementing Servlets

You can define simple web servlets inside your PolicyCenter application. You can define extremely simple HTTP entry points to custom code using this approach. These are separate from web services that use the SOAP protocol. These are separate also from the Guidewire PCF entrypoints feature.

Use this technique to define arbitrary Gosu code that a user or tool can call from a configurable URL. You can define a Gosu block that can determine from the URL whether your servlet owns the HTTP request.

Servlets provide no built-in object serialization or deserialization, such as is done in the SOAP protocol.

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 it.

Extending the base class `javax.servlet.http.HttpServlet` provides no inherent security for the servlet. By default users can trigger servlet code without authenticating. This is dangerous in a production system. PolicyCenter includes a utility class called `gw.servlet.ServletUtils` that you can use in your servlet to require authentication.

WARNING You must add your own authentication system for servlets to protect information and data integrity. Carefully read “[Implementing Servlet Authentication](#)” on page 603. If you have any questions about server security, contact Guidewire Customer Support.

Creating a Basic Servlet

To create a basic servlet

1. Write a Gosu class that extends the class `javax.servlet.http.HttpServlet`.

2. Register the servlet class in the file:

PolicyCenter/configuration/config/servlets.xml

Add one <servlet> element that references the fully qualified name of your class, for example:

```
<servlets xmlns="http://guidewire.com/servlet">
  <servlet class="com.example.servlets.MyServletClass"/>
</servlets>
```

3. Add the @Servlet annotation on the line before your class definition.

You must pass arguments to the constructors to specify which URLs your servlet handles. There are multiple versions of the constructors for different use cases.

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". In other words, your servlet gets the servlet URL substring identified as *YOUR_SERVLET_STRING* in the following URL syntax:

- `@Servlet(pathPattern : String)`

Use this annotation constructor syntax for high-performance pattern matching, though 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 filenames with limited wildcard support for ? (single character match) and * (multiple character match) similar to DOS filename 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 pass a Gosu block that can do arbitrary matching on the servlet query path. Performance depends greatly on what you do in your block. As a parameter in parentheses for the annotation, pass a Gosu block that takes a URL String. Write the block such that the block returns `true` if and only if the user URL matches what your servlet handles.

If you only use the `startsWith` method of the `String` argument, for example

`path.startsWith("myprefix")`, the servlet URL checking code is high performance.

You could do more flexible pattern matching with other APIs, such as full regular expressions. Using regular expressions lowers performance for high volumes of servlet calls. The following example uses regular expressions with the `matches` method on the `String` type:

```
@Servlet(\ path : String ->path.matches("/test(/.*)?"))
```

This example servlet responds to URLs that start with the text "/test" in the servlet query path, and optionally a slash followed by other text.

4. Override the `doGet` method to do your actual work. Your `doGet` method takes a servlet request object (`HttpServletRequest`) and a servlet response object (`HttpServletResponse`).

5. In your `doGet` method, determine what work to do using the servlet request object.

WARNING You must add your own authentication system for servlets to protect information and data integrity. Carefully read "Implementing Servlet Authentication" on page 603. If you have questions about server security, contact Guidewire Customer Support.

Important properties on the request object include:

- `RequestURI` – Returns the part of this request's URL from the protocol name up to the query string in the first line of the HTTP request.
- `RequestURL` – Reconstructs the URL the client used to make the request.
- `QueryString` – Returns the query string that is contained in the request URL after the path.
- `PathInfo` – Returns any extra path information associated with the URL the client sent when it made this request.

- **Headers** – Returns all the values of the specified request header as an Enumeration of String objects.

For full documentation on this class, refer to these Sun Javadoc pages:

<http://docs.oracle.com/javaee/6/api/javax/servlet/http/HttpServletRequest.html>
<http://docs.oracle.com/javaee/6/api/javax/servlet/http/HttpServletResponse.html>

6. In your doGet method, write an HTTP response using the servlet response object. For example, the following simple response sets the content MIME type and the status of the response (OK):

```
resp.ContentType = "text/plain"
resp.setStatus(HttpServletResponse.SC_OK)
```

To write output text or binary data, use the `Writer` property of the response object. For example:

```
resp.getWriter.append("hello world output")
```

For example, the following simple Gosu servlet works although it provides no authentication:

```
package mycompany.test

uses gw.servlet.Servlet
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 doGet(req: HttpServletRequest, response: HttpServletResponse) {

        // ** SECURITY WARNING - FOR REAL PRODUCTION CODE, ADD AUTHENTICATION CHECKS HERE!

        // Trivial servlet response to test that the servlet responds
        response.setContentType = "text/plain"
        response.setStatus(HttpServletResponse.SC_OK)
        response.getWriter.append("I am the page " + req.PathInfo)
    }
}
```

Add an element in the `servlets.xml` file for your new servlet class. Run the QuickStart server at the command prompt: `gwpc dev-start` and test the servlet at the URL:

<http://localhost:PORT/pc/service/test>

Change the port number as appropriate for your application.

For a production servlet, you must add authentication. See “[Implementing Servlet Authentication](#)” on page 603.

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. This is dangerous in a production system, generally speaking. PolicyCenter includes a utility class called `gw.servlet.ServletUtils` that you can use in your servlet to require authentication.

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.

There are three methods in the `ServletUtils` class, each of which corresponds 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 `doWork`.

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 into 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. For example:</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 ServiceToken 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>Even 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 style 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/ login name pairs	<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 username and password and call this method. This method forces creation of a new session.</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>

You can combine the use of multiple methods of `ServletUtils` in your code.

For example, 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`, attempt to use HTTP Basic authentication by calling the method `getBasicAuthenticatedUser`.

The following code demonstrates this technique in the doGet method and calls to a separate method to do the main work of the servlet.

```
package mycompany.test

uses gw.servlet.Servlet
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 doGet(req: HttpServletRequest, response: HttpServletResponse) {
        //print("Beginning call to doGet()...")

        // 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(HttpServletResponse.SC_UNAUTHORIZED,
                    "Unauthorized. HTTP Basic authentication error.")
                return // Be sure to RETURN early because authentication failed!
            }

        if (user == null) {
            response.sendError(HttpServletResponse.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.RequestURI: '${req.RequestURI}'\n" +
            "authenticated user name: '${user.DisplayName}'\n"

        // debugging in the console
        //print(responseText)

        // for output response
        response.ContentType = "text/plain"
        response.setStatus(HttpServletResponse.SC_OK)
        response.getWriter.append(responseText)
    }
}
```

To test session authentication servlet code

1. In Studio, create the `mycompany.test.TestingServlet` as shown earlier in this topic.
2. Register this servlet in `servlets.xml`.
3. Run the QuickStart server at the command prompt: `gwpc dev-start`
However, do not yet log into the PolicyCenter application.
4. Test the servlet with no authentication by going to the URL:
`http://localhost:PORT/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.
```

5. Log into the PolicyCenter application with valid credentials.

6. Test the servlet with no authentication by going to the URL:

```
http://localhost:PORT/pc/service/test
```

You see a message:

```
REQUEST SUCCEEDED  
req.RequestURI: '/cc/service/test'  
req.PathInfo: '/test'  
req.RequestURI: '/cc/service/test'  
authenticated user name: 'Super User'
```

For testing of the HTTP Basic authentication, sign out of the PolicyCenter application to remove the session context. Next, re-test your servlet and use a tool that supports adding HTTP headers for HTTP Basic authentication.

Alternative APIs for Authentication

Guidewire recommends using the `ServletUtils` APIs for managing authentication. See “[Implementing Servlet Authentication](#)” on page 603. If you use `ServletUtils`, you can use the session key if available and if not you can use HTTP Basic authentication headers or custom headers.

An alternative approach for authentication is to use the legacy class `AbstractGWAAuthServlet` to translate the security headers in the request and authenticate with the Guidewire server. There is a subclass called `AbstractBasicAuthenticationServlet`, which authenticates using HTTP Basic authentication. You can view the source code to both classes in Studio. These classes are provided primarily for legacy use because they do not support using more than one type of authentication at run time.

Abstract Guidewire Authentication Servlet 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 `AbstractGWAAuthServlet`. You must override the following methods:

- `doGet` – Your main task is to override the `doGet` method to do your main work. PolicyCenter already authenticates the session key if it exists before calling your `doWork` method. For more information about the parameters in this method, see “[Creating a Basic Servlet](#)” on page 601.
- `authenticate` – Create and return a session ID
- `storeToken` – You can store the session token in this method, but you can also 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 )  
}
```

Abstract HTTP Basic Authentication Servlet Class

The `AbstractBasicAuthenticationServlet` class extends `AbstractGWAAuthServlet` to support HTTP Basic authentication.

Your main task is to override the `doGet` method to do your main work. PolicyCenter already authenticates the using HTTP Basic authentication headers before calling your `doWork` method. For more information about the parameters in this method, see “[Creating a Basic Servlet](#)” on page 601.

Also, override the `isAuthenticationRequired` method and return `true` if authentication is required for this request.

The following example responds to servlet URL substrings that start with the string `/test/`. If an incoming URL matches that pattern, the servlet simply echoes back the `PathInfo` property of the response object, which contains the path.

```
package mycompany.test

uses gw.servlet.HttpServlet
uses javax.servlet.http.HttpServletRequest
uses javax.servlet.http.HttpServletResponse
uses javax.servlet.http.HttpServlet
uses gw.api.util.Logger

@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 true/false if user authentication is required
        // -----
        return true;
    }
}
```

This servlet responds to URLs with the word `test` in the service query path, such as the URL:

`http://localhost:8080/pc/service/test/is/this/working`

To test this, you must use a tool that supports adding HTTP Basic authentication headers for the username and password.

Note that the text `"/test"` in the URL is the important part that matches the servlet string. Change the port number and the server name to match your application.

Your web page displays the following:

`I am the page /test/is/this/working`

Use this basic design pattern to intercept any of the following:

- A single page URL
- An entire virtual file hierarchy, as shown in the previous example
- Multiple page URLs that are not described in traditional file hierarchies as a single root directory with subdirectories. For example, you could intercept URLs with the regular expression:

`"(/.*)?/my_magic_subfolder_one_level_down"`

That would match all of the following URLs:

```
http://localhost:8080/pc/service/test1/my_magic_subfolder_one_level_down
http://localhost:8080/pc/service/test2/my_magic_subfolder_one_level_down
http://localhost:8080/pc/service/test3/my_magic_subfolder_one_level_down
```


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.

This topic includes:

- “Why Gosu Templates are Useful for Data Extraction” on page 609
- “Data Extraction Using Web Services” on page 610

Why Gosu Templates are Useful 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. See “Gosu Templates” on page 353 in the *Gosu Reference Guide*.

Possible output formats include:

- 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 using templates are as follows:

- **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. See “Servlets” on page 601. 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 let you address each integration point in your network. Follow these 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.

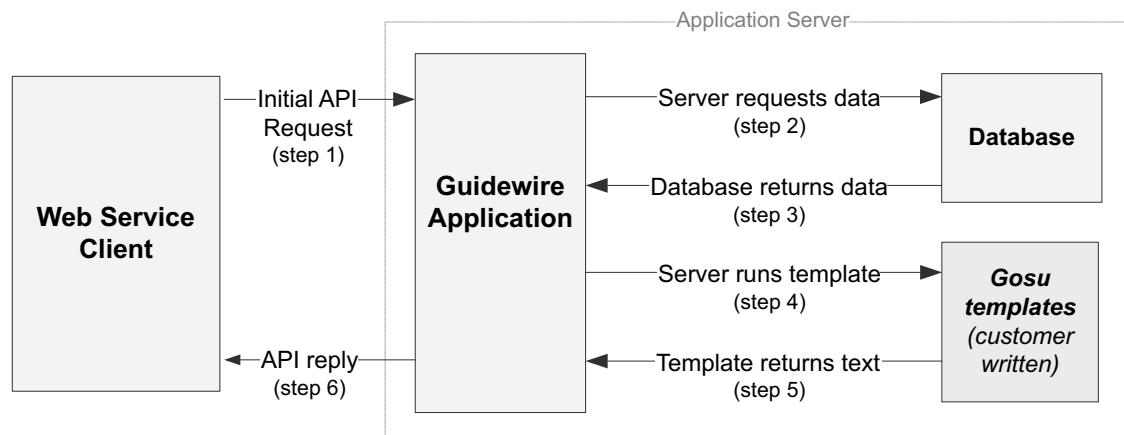
For more information, see “Data Extraction Using Web Services” on page 610.

Data Extraction Using Web Services

You can write your own web services that use Gosu templates to generate results. For more information about writing Gosu templates, see “Gosu Templates” on page 353 in the *Gosu Reference Guide*.

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:

- 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.
```

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:

```
<%= (parameters.get("xyz") == null)? "NO VALUE!" : parameters.get("xyz") %>
```

Built in Templates

There are example Gosu templates included with PolicyCenter. Refer to the files within the PolicyCenter deployment directory:

```
PolicyCenter/modules/configuration/config/templates/dataextraction/*.gst
```

Using Loops in Templates

Gosu templates can include loops that iterate over multiple objects such as all policy revision associated with a policy. For example, you could use a Gosu template with template code with the root object `policy` might look something like this:

```
<% for (var thisPolicyRev in policy.revisions) { %>
    Policy revision number: <%= thisPolicyRev.PolicyNumber %>
<% } %>
```

This template might generate something like:

```
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 Common for Integration

Gosu Libraries

You can access Gosu libraries from within templates similar to how you would access them from within Guidewire Studio, using the `Libraries` object:

```
<%= Libraries.Financials.getTotalIncurredForExposure(exposure) %>
```

Java Classes Called From Gosu

Just like any other Gosu code, your Gosu code in your template use Java classes.

```
var myWidget = new mycompany.utils.Widget();
```

See also

- For information about calling Java from Gosu, see “Calling Java from Gosu” on page 123 in the *Gosu Reference Guide*.

Logging

You can send messages to the PolicyCenter log file using the PolicyCenter logger object. Access the logger from Gosu using the following code:

```
libraries.Logger.logInfo("Your message here...")
```

Typecode Alias Conversion

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.

See also

- For more information on the Gosu utility class, see “Using Gosu or Java to Translate Typecodes” on page 94.

Logging

PolicyCenter provides a robust logging system based on the open source Apache log4j project. The log4j system flexibly outputs log statements with arbitrary granularity at a specific logging level that indicate what to write to files. You can configure the logging system fully at runtime by using external configuration files. You can use the PolicyCenter logging system throughout your application code. However, special features of the logging system are particularly useful for integration developers, and this topic discusses them.

This topic includes:

- “Logging Overview For Integration Developers” on page 615
- “Logging Properties File” on page 616
- “Logging APIs for Java Integration Developers” on page 617

See also

“Configuring Logging” on page 23 in the *System Administration Guide*

Logging Overview For Integration Developers

Logging Elements

The logging system in PolicyCenter comprises these elements:

- **Logging properties files** – A `logging.properties` file specifies what information to log. For instance, you can choose to log absolutely everything relevant to integration, log plugin information, or log nothing. You can choose a large set of logging properties that log certain errors for certain cases or warnings for other cases. The logging properties file uses the format for the Java tool log4j.
- **Log files** – Create log files anywhere accessible from the server. You can create different types of log messages for different contexts in different files or even multiple server directories.
- **Code that triggers logging messages** – Many built-in parts of PolicyCenter can log information, warnings, errors, or arbitrary debugging information. As an integration developer, your code can log anything you want. However, code that triggers logging does not force log messages to be written to files. The logging properties file specifies what PolicyCenter actually does with any specific logging information.

Logging Types: Category-based and Class-based

PolicyCenter provides two ways to configure logging:

- **Category-based logging** – PolicyCenter provides a hierarchical, abstract category system for logging that avoids dependencies on specific Java classes. Category-based logging operates independently of Java packages and supports quick and easy configuration of log levels, log file locations, and other logging settings. Guidewire strongly recommends that you use category-based logging.
- **Class-based logging** – If you have legacy Java code that uses log4j class-based logging, it might be easier to continue your use class-based logging. However, Guidewire strongly recommends that you migrate your log4j code to category-based logging so your code integrates with the pre-defined logging categories of PolicyCenter.

Logging Properties File

The logging properties file (`logging.properties`) configures what to log and where to log it. The logging properties file is in the log4j format, which the Apache project defined for the log4j system. This topic only briefly mentions the details of how to configure log4j files such as these.

Each logging properties file is separated into blocks such as the following example:

```
# set logging levels for the category: "log4j.category.Integration.plugin"
log4j.category.Integration.plugin=DEBUG, PluginsLog
# To remove logging for that category, comment out the previous line with an initial "#"

# The following lines specify how to write the log, where to write it, and how to format it
log4j.additivity.PluginsLog=false
log4j.appenders.PluginsLog=org.apache.log4j.DailyRollingFileAppender
log4j.appenders.PluginsLog.File=C:/Guidewire/PolicyCenter/logs/plugins.log
log4j.appenders.PluginsLog.DatePattern = .yyyy-MM-dd
log4j.appenders.PluginsLog.layout=org.apache.log4j.PatternLayout
log4j.appenders.PluginsLog.layout.ConversionPattern=%-10.10X{server} %-4.4X{userID} %d{ISO8601} %p %m%n
```

In this example, the following line defines the class, log level, and a log name (`AdaptersLog`):

```
log4j.category.Integration.plugin=DEBUG, AdaptersLog
```

This line specifies to use a file appender (a standard local log file):

```
log4j.appenders.PluginsLog=org.apache.log4j.DailyRollingFileAppender
```

The line after that specifies the location of the log:

```
log4j.appenders.PluginsLog.File=C:/Guidewire/PolicyCenter/logs/plugins.log
```

See also

<http://logging.apache.org/log4j/1.2/index.html>

Logging Categories for Integration

Logging categories are hierarchical. For example, enabling a log file appender for the category `log4j.category.plugin` enables `log4j.category.plugin.IValidationAdapter`. If there are subcategories defined for that category, those are enabled also. If you use logging categories from Java code, use the `LoggerCategory` class, which includes several static instances of the `Logger` interface that are pre-defined for common use.

See also

- “Category-based Logging” on page 617.
- For a current list of logging categories, see “Understanding Logging Categories” on page 25 in the *System Administration Guide*

Logging APIs for Java Integration Developers

Category-based Logging

PolicyCenter provides an API to the log4j-based logging system by using the `LoggerCategory` class. The `LoggerCategory` class contains predefined static instances of the `Logger` interface. The `LoggerCategory` class is available to your Java code and your web services API client code.

For your Java plugins, logger configuration is automatic because the server already instantiated and configured a *logger factory*. A logger factory is the object that configures what to log and where to log it. A Java plugin automatically inherits the logging properties of the application server.

See also

If you want to use logging within a Java plugin, see “[Logger Classes](#)” on page 617.

See also

The API Reference Javadoc for `gw.api.system.logging.LoggerFactory`

Logger Classes

To use the `LoggerCategory` class, you first need an instance of the `Logger` class. The easiest way to get an instance is to use the static instances of this class predefined for common top level loggers, such as `PLUGIN` and `API`. You can access these loggers as properties of the `LoggerCategory` class. For instance, `LoggerCategory.PLUGIN` refers to the static instance of the `Logger` interface for plugins.

For example, you can use code like the following to log an error.

```
LoggerCategory.PLUGIN.error(Document + ", missing template: " + stringWithoutDescriptor);
```

To use this tool within a Java class, declare a private class variable of interface `Logger` like this.

```
private Logger _logger = null;
```

Then at runtime, get an instance of the `Logger` object. For example, use the built-in static instances of the `LoggerCategory` class.

```
_logger = LoggerCategory.PLUGIN;
```

You can now write to this logger object with `Logger` methods. The methods you typically use most are methods that log a message at a specific log4j logging level: `info`, `warn`, `trace`, `error`, and `debug`.

The following example logs a message at the `INFO` logging level.

```
_logger.info("Setting up logger for MySpecialCode...");
```

The logger does not append this message to a file unless the logger is configured to do so. You must set logging properties to enable that logging level and define a filename path for the log file output.

PolicyCenter uses the initial setup of the logging factory to determine the logging level for this logger based on its category. For plugins, it inherits the server settings. However, you can override the logging level by changing `logging.properties`. You can also temporarily adjust the logging level for a logger by using the [Set Log Level](#) screen in PolicyCenter. Changes made with the [Set Log Level](#) screen only persist until the PolicyCenter server is restarted.

If you want to use a logger for which there is no static instance, use one of two alternate calls to the `LoggerFactory` class. The most common approach is to create a logger as a sublogger of an existing logger.

```
Logger logger = LoggerFactory.getLogger(LoggerCategory.PLUGIN, "IApprovalAdapter");
logger.info("My info message here")
```

Alternatively, create a new root category.

```
LoggerCategory logger = LoggerFactory.getLogger("MyRootCategoryName");
logger.info("My info message here")
```

To configure your new category in `logging.properties`, define the new logger and give it its own appender, as the following example shows.

```
log4j.category.IApprovalAdapter=DEBUG, MyLog
log4j.additivity.MyLog=false
log4j.appender.MyLog=org.apache.log4j.DailyRollingFileAppender
log4j.appender.MyLog.File=c:/gwlogs/messaging.log
log4j.appender.MyLog.DatePattern = .yyyy-MM-dd
log4j.appender.MyLog.layout=org.apache.log4j.PatternLayout
log4j.appender.MyLog.layout.ConversionPattern=%-10.10X{server} %-4.4X{userID} %d{ISO8601} %p %m%n
```

If you created a new root category, replace `IApprovalAdapter` in the example above with your new category.

See also

- “Understanding Logging Levels” on page 25 in the *System Administration Guide*
- “Set Log Level” on page 150 in the *System Administration Guide*

Class-based Logging (Not Generally Recommended)

Instead of using abstract logging categories to identify related code, you can use the fully-qualified name of a class. By using a fully qualified class name, The class name and package define the hierarchy you use to define logging configuration settings.

IMPORTANT Guidewire strongly recommends you use the category-based approach, instead of the class-based approach described in this topic.

For instance, instead of your plugin code writing log messages with `LoggerCategory` to the `log4j.category.Integration.plugin.IValidationAdapter` class, configure a logger based on the actual class of your plugin. For example, you might use `com.mycompany.myadapters.myValidationAdapter`. To use the class-based approach, use the `Logger` and `LoggerFactory` classes.

Just as for category-based logging, plugin logger configuration is automatic because the server already instantiated and configured a *logger factory*. The logger factory configures what to log and where to log it. However, for web services API client code, you must explicitly set up the logger factory using the `LoggerFactory` class. Plugin code and web services API client code can set up a logger factory using the `LoggerFactory` class.

To use class-based logging in your code

1. Configure the logger in the `logging.properties` file by using the class name instead of the category name.
2. Use a `LoggerFactory` instance to create a `Logger` instance.
3. Send logging messages to that `Logger` instance.

For a typical example for a plugin, set a class private variable.

```
private Logger _logger = null;
```

Then in your set up code, initialize the logger.

```
_logger = LoggerFactory.getLogger(MyJavaClassName.class);
```

And then you can send logger messages with it, with similar methods as in `LoggerCategory`.

```
_logger.info("Setting up logger ...");
```

See also

- “Category-based Logging” on page 617

Dynamically Changing Logging Levels

You can change logging levels without redeploying PolicyCenter. For more information, see “Making Dynamic Logging Changes without Redeploying” on page 30 in the *System Administration Guide*.

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.

This topic includes:

- “Proxy Server Overview” on page 621
- “Configuring a Proxy Server with Apache HTTP Server” on page 622
- “Certificates, Private Keys, and Passphrase Scripts” on page 623
- “Proxy Server Integration Types for PolicyCenter” on page 623
- “Proxy Building Blocks” on page 624

See also

- For configuring web services URLs to support proxy servers, see “Working with Web Services” on page 137 in the *Configuration Guide*.
- For general information about web services, see “Web Services Introduction” on page 37.

Proxy Server Overview

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.

If a proxy server handles incoming connections from an external Internet service to PolicyCenter and not just outgoing requests from PolicyCenter, some people would call it a *reverse proxy server*. For the sake of simplicity, this topic refers simply to a 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/2.0/ssl/ssl_intro.html
Overview of Apache's SSL module	http://httpd.apache.org/docs/2.0/mod/mod_ssl.html
Overview of Apache's proxy server module	http://httpd.apache.org/docs/2.0/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.

Be aware 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, then directly connect to the remote service or through a testing-only proxy server. In contrast, if running in the production environment, then always connect through the official proxy server.

See also

- For more information, see “Working with Web Services” on page 137 in the *Configuration Guide*.
- For additional information about web services, see “Web Services Introduction” on page 37.

Configuring a Proxy Server with Apache HTTP Server

The 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 then 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.

Apache Basic Installation Checklist

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.

Follow these high-level steps for installing Apache

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.

IMPORTANT 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
<code>\$ReverseProxyTrustedProtectedPrivateKeyFile</code>	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, <code>\$ReverseProxyTrustedPassPhraseScript</code>

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.

To configure the Bing Maps geocoding plugin to use a proxy server

1. In PolicyCenter 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
 - b. In the Web Service Collection editor, click **Add Setting** and then **Override URL**.
 - c. In the **Override URL** field, enter your proxy server URL and port number for the Bing Maps web service.
2. In the configuration of your Apache proxy server, add a configuration building block for the Bing Maps URL. Follow the pattern for configuration building blocks described in “Downstream Proxy With No Encryption” on page 625

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. Simply 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 Contact Manager Studio with the overriding URL of your proxy server.

See also

- For more information on how to configure WS-I web services for proxy servers, see “Working with Web Services” on page 137 in the *Configuration Guide*.
- For more information about the geocoding plugin, see “Geographic Data Integration” on page 229.

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 following Apache configuration building block:

- “Upstream (Reverse) Proxy with Encryption for User Connections” on page 626

See also

- For more information about SSL encryption, see the “Securing PolicyCenter Communications” on page 91 in 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, instead of:

```
Listen $PROXY_PORT_NUMBER_HERE
```

Replace it with:

```
Listen 1234
```

...to listen on port 1234.

IMPORTANT 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.

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 this 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 this Apache configuration building block:

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

#Disable forward proxying for security purposes
ProxyRequests Off

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

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

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

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

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

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

#Logs redirected to appropriate location
ErrorLog $ApacheErrorLog

</VirtualHost>
```

Upstream (Reverse) Proxy with Encryption for Service Connections

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

Use this 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.

Edit your proxy server configuration so it is aware of:

- 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)

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 an additional 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>

```

You must substitute the following parameters shown in the snippet:

<i>port</i>	Specifies the port number for the additional connector for access through the proxy.
<i>proxyName</i>	Identifies the deployment server's name.
<i>proxyPort</i>	Specifies the port for encrypted access through Apache.
<i>scheme</i>	Identifies the protocol used by the client to access the server.

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

See also

- For more information about SSL encryption, see “Securing PolicyCenter Communications” on page 91 in the *System Administration Guide*.

Java and OSGi Support

This topic describes ways to write and deploy Java code in PolicyCenter, including accessing entity data from Java. For example, you could implement PolicyCenter plugin interfaces 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 helps cleanly isolate code modules and any necessary Java libraries. Guidewire recommends OSGi for all new Java plugin development.

See also

- You can write Gosu code that uses Java types. See “Calling Java from Gosu” on page 123 in the *Gosu Reference Guide*.
- For an introduction to Gosu from a Java perspective, see “Gosu Introduction” on page 15 in the *Gosu Reference Guide*.

This topic includes:

- “Overview of Java and OSGi Support” on page 629
- “Accessing Entity and Typecode Data in Java” on page 633
- “Accessing Gosu Classes from Java Using Reflection” on page 644
- “Gosu Enhancement Properties and Methods in Java” on page 645
- “Class Loading and Delegation for non-OSGi Java” on page 645
- “Deploying Non-OSGi Java Classes and JARs” on page 646
- “OSGi Plugin Deployment with IntelliJ IDEA with OSGi Editor” on page 647
- “Advanced OSGi Dependency and Settings Configuration” on page 654
- “Updating Your OSGi Plugin Project After Product Location Changes” on page 655

Overview of Java and OSGi Support

You can deploy Java code within PolicyCenter. There are several different ways you can use Java.

Typical customers write Java code primarily to implement PolicyCenter plugin interfaces.

However, you can 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 PolicyCenter Studio. It is unsupported to add or modify Java class files in the PolicyCenter Studio user interface. Although PolicyCenter Studio does not hide user interface tools that add Java classes to the file hierarchies, those features are unsupported.

If you are deploying Java plugins or OSGi plugins, carefully read about your IDE options in “[Implementing Plugin Interfaces in Java and Optionally OSGi](#)” on page 630.

Learning More About Entity Java APIs

If you want to deploy Java code and you do not use Guidewire entity data, the main thing you need to know is where to put Java classes and libraries. See “[Deploying Non-OSGi Java Classes and JARs](#)” on page 646.

If your Java code needs to get, set, or query Guidewire entity data, you must also understand how PolicyCenter works with entity data in Java. See “[Accessing Entity and Typecode Data in Java](#)” on page 633.

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 is not typesafe, which means you cannot catch some types of errors at compile time. For example, you must use reflection to access the following from Java: Gosu classes, Gosu interfaces, and Gosu enhancements. See the following topics:

- “[Accessing Gosu Classes from Java Using Reflection](#)” on page 644
- “[Gosu Enhancement Properties and Methods in Java](#)” on page 645

Note that some of the supported PolicyCenter types that you might use in Gosu are actually implemented in Java and thus require no special access from Java.

Implementing Plugin Interfaces in Java and Optionally OSGI

Many customers write Java code primarily to implement plugin interfaces. For general information about plugins, see “[Plugin Overview](#)” on page 123.

Conceptually, there are two main steps to implement a plugin:

1. **Write a class that implements the interface** – See “[Implementing Plugin Interfaces](#)” on page 124
2. **Register your plugin implementation** – See “[Registering a Plugin Implementation Class](#)” on page 126

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. 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. For details of deploying the files, see “[Deploying Non-OSGi Java Classes and JARs](#)” on page 646.

IMPORTANT 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.

For more information about the OSGi standard, refer to:

<http://www.osgi.org/Technology/WhyOSGi>

The most important benefits of OSGi for PolicyCenter plugin development are:

- 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 startup time.
- 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 the set of database rows to update. See “Bundles and Database Transactions” on page 337 in the *Gosu Reference Guide*.
- In the OSGi standard, *bundle* refers to a registered OSGi component. PolicyCenter documentation relating to Java and plugins sometimes refers to *OSGi bundles*.

IDE Options for Plugin Development in the Java Language

You can use any Java IDE that is separate from PolicyCenter Studio. It is unsupported to edit Java directly in Studio.

If you are writing an OSGi plugin implementation, there are 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 than PolicyCenter 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)
PolicyCenter Studio	Yes	--	--
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.

Inspections to Flag Unsupported Internal Java 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 your 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 of your own choice	Optional installation. For compatibility with specific IntelliJ IDEA versions, please contact Guidewire Customer Support. See “Installing the Internal API Inspection in Your Own Instance of IntelliJ IDEA” on page 632
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 @gw.lang.InternalAPI.
PolicyCenter Studio	WARNING: PolicyCenter Studio does not support Java coding directly in the IDE. Do not create Java classes directly in PolicyCenter Studio. If you want to code in Java, you must use a separate IDE for Java development. See “IDE Options for Plugin Development in the Java Language” on page 631

Installing the Internal API Inspection in Your Own Instance of IntelliJ IDEA

The recommended approach for Java development is with the included IntelliJ IDEA with OSGi Editor. If you instead choose to use your own separate instance of IntelliJ IDEA, you may be able to use the Internal API inspection, depending on your version of IntelliJ IDEA. For compatibility with specific IntelliJ IDEA versions, please contact Guidewire Customer Support.

IMPORTANT The IntelliJ IDEA with OSGi Editor automatically 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.

To install the IntelliJ IDEA Internal API Code Inspection

1. Open your separate instance of IntelliJ IDEA.
2. Navigate to **Settings → Plugins**.
3. Click **Install from disk...**
4. Navigate to the directory:
PolicyCenter/studio/configstudio/internal-api-idea-plugin/lib/
5. Select the JAR in that directory.
6. Let IntelliJ IDEA restart after installing the plugin.
7. Navigate to **Settings → Inspections**.
8. Expand the node **Portability Issues**.
9. Check the box **Use of internal API**.
10. Click **OK**.

Accessing Entity and Typecode Data in Java

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

You can write Java code that accesses entity data from:

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

After you call a script that regenerates PolicyCenter Java API libraries, you can write Java code that uses them to access entity data. See “Regenerating Integration Libraries and WSDL” on page 31.

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.

IMPORTANT Do not create Java classes directly in Studio. To code in Java, you must use a separate IDE for Java development. For example, use 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 through the Java API library interface has data getter and setter methods to get and set data. 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. See “Plugin Overview” on page 123.
- **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.

In both cases, your code can:

- 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:

- For non-OSGi Java, see “Deploying Non-OSGi Java Classes and JARs” on page 646
- For OSGi plugins, see “Using Third-Party Libraries in Your OSGi Plugin” on page 651s

Regenerating Java API Libraries

Anytime 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.

To regenerate the Java API libraries

1. In Windows, open a command prompt.
2. Change your working directory with the following command:

```
cd PolicyCenter/bin
```

- Use the `regen-java-api` command:

```
gwpc regen-java-api
```

This command generates Java entity interfaces and typelist classes in libraries in the location:

```
PolicyCenter/java-api/lib
```

This command generates Javadoc documentation in the location:

```
PolicyCenter/java-api/doc
```

Entity Packages and Customer Extensions from Java

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

In the PolicyCenter Java API, you can reference a type directly by its fully-qualified name. However, for PolicyCenter entity types, from Java the fully-qualified name of an entity is not `entity.ENTITYNAME` nor simply the entity name. The syntax `entity.ENTITYNAME` or using the entity name with no package is a shortcut of the Gosu type system.

If you want only the base configuration properties, the type name is the same in Java as in Gosu but the package varies by entity type. Some aspects of the fully-qualified names of the interfaces are configurable. See “Configuring Entity and Typelist Fully-Qualified Names from Java” on page 637. PolicyCenter exposes each entity type as several interfaces, which the following table summarizes.

Terminology	Description	When it exists	Entity name suffix	For more information, including the Java package for the interface
Entity types that Guidewire originally creates				
<i>base entity interface</i>	Contains only the base configuration properties.	All entity types have a base entity interface.	n/a	“Accessing Entity Properties and Methods With Base and Core Extension Interfaces” on page 635
Entity types that you originally create				
<i>customer extension entity interface</i>	Includes customer data model extension properties. If a core extension interface exists for that entity type, the customer extension interface extends from the core extension entity interface. Otherwise, the customer extension interface extends from the base entity interface.	Exists only if there are customer data model extensions.	Ext	“Customer Extension Entity Interface” on page 636
<i>customer entity interface</i>	Contains all properties	All entities that you create have this interface	n/a	“Entity Interfaces for Completely Custom Entity Types” on page 637.

Accessing Entity Properties and Methods With Base and Core Extension Interfaces

From Java, the base entity interface package includes a *prefix* and a *subpackage* that defines a general area of functionality in the application. The general pattern for the fully-qualified base entity type name is:

```
PACKAGE_PREFIX.SUBPACKAGE.entity.ENTITYNAME
```

For entity types that multiple Guidewire applications share, the pattern is:

```
gw.p1.SUBPACKAGE.entity.ENTITYNAME
```

For entities specific to the PolicyCenter application, the pattern is:

```
gw.pc.SUBPACKAGE.entity.ENTITYNAME
```

For example, fully-qualified name of the `Address` entity is:

```
gw.p1.contact.entity.Address
```

Instead of memorizing fully-qualified names of the entities, use the auto-completion features of your IDE. For example, if you type `Address` then type `CTRL+Space` in IntelliJ IDEA, the IDE presents options including the entity with the pattern described above. After you choose the correct entity type, the IDE inserts an `import` statement at the top of the file. For example, the IDE might add the following line:

```
import gw.p1.contact.entity.Address;
```

If you write Java code that implements a PolicyCenter plugin interface, interface method arguments and return types that reference entity types always uses the Java name for the type. The Java fully-qualified name for the entity type is different than the type name in Gosu, and in fact in Java is represented by three interfaces. Remember this difference as you review documentation examples that may show code in Gosu.

Some aspects of the fully-qualified names of the interfaces are configurable. See “Configuring Entity and Type-list Fully-Qualified Names from Java” on page 637.

The Java entity interfaces are supported only in Java code. From Gosu, use the syntax `entity.ENTITYNAME`, or where unambiguous just type the entity name.

Core Extension Entity Interfaces

For some entity types, the Guidewire platform defines a base version and then PolicyCenter extends the base entity interface with properties and methods for application-specific business logic.

Only for such entities, PolicyCenter adds a *core extension entity interface* with the suffix `CoreExt`. For example, the `Message` entity contains many properties in its base entity interface. However, each Guidewire application adds additional properties to `Message` for business logic unique to each application to reference a primary object for that message.

The package for the core extension entity interface is slightly different from the base entity interface. For entities that have core extension entity interfaces, the base entity interface has the pattern:

```
gw.p1.SUBPACKAGE.entity.ENTITYNAME
```

By definition, core extension entity interfaces are specific to the PolicyCenter application. Therefore, the pattern matches the pattern for application-specific entities:

```
gw.pc.SUBPACKAGE.entity.ENTITYNAMECoreExt
```

For example, Java code that references the base entity interface and core extension entity interface for the `Message` entity has the following lines at the top of the Java file:

```
import gw.p1.messaging.entity.Message;
import gw.pc.messaging.entity.MessageCoreExt;
```

The core extension entity interface extends the base entity interface, so it also contains all the methods of the base entity interface. To use application-specific properties, downcast an object reference to the core extension entity interface.

For example, in a PolicyCenter message transport plugin, to use the `Message.Policy` property, you must downcast to the *core extension interface* to access that property:

```
import gw.pc.claim.entity.Policy;
import gw.p1.messaging.entity.Message;
```

```

import gw.pc.messaging.entity.MessageCoreExt;
class MyMessageTransport implements MessageTransportPlugin {
    // ...
    public void send(Message message, String transformedPayload) {
        // IMPORTANT: To access some properties, cast Message to subinterface MessageCoreExt
        MessageCoreExt m = (MessageCoreExt) message;

        // access a property on MessageCoreExt that is absent on Message
        Policy p = m.getPolicy();

        // ... use this Policy information and send a message...
    }
}

```

Alternatively, to access the application-specific properties or methods, you can use the customer extension entity interface. Only if there are customer data model extensions, PolicyCenter creates a customer extension entity interface, which extends from the base entity interface or the core extension interface if it exists. For details, see “Customer Extension Entity Interface” on page 636.

The Java entity interfaces are supported only in Java code. From Gosu, use the syntax `entity.ENTITYNAME`, or where unambiguous just type the entity name.

Customer Extension Entity Interface

For each entity, if there are any customer data model extensions, there is an additional interface called the *customer extension entity interface*. If there are no customer data model extensions, PolicyCenter does not create this interface.

Important qualities of the customer extension entity interface:

- The interface contains your data model extension properties.
- The interface name has the suffix `Ext`.
- The interface package is different from the package of the base entity interface. The package prefix changes to the *Java API package prefix* as defined in the file `extensions.properties`. See the table in “Configuring Entity and Typelist Fully-Qualified Names from Java” on page 637.
- The customer extension entity interface extends from the base entity interface or the core extension interface if it exists. Therefore if there is a customer extension entity interface, it contains the complete set of all possible properties defined by data model configuration files.
- Like all the other Java entity interfaces, the customer extension entity interface does not include properties added by Gosu enhancements. See “Gosu Enhancement Properties and Methods in Java” on page 645.

The Java fully-qualified name of the customer extension entity interface is:

`PACKAGE_PREFIX.SUBPACKAGE.entity.ENTITYNAMEExt`

By default, `PACKAGE_PREFIX` is `extensions.pc`, and the subpackage varies by entity type in the entity declaration. For configuration of the `PACKAGE_PREFIX`, see “Configuring Entity and Typelist Fully-Qualified Names from Java” on page 637.

For example, consider the `Address` entity type. In Gosu, use either of the following syntax styles:

```

Address
entity.Address

```

The Java fully-qualified name of the base entity interface is:

`gw.pl.contact.entity.Address`

By default, the Java fully-qualified name of the customer extension interface is:

`extensions.pc.contact.entity.AddressExt`

If the customer extension entity interface exists, you can downcast a variable declared to the base entity interface to the customer extension entity class. The customer extension class contains the customer extension properties as well as all the properties and methods from the base and core extension entity interfaces.

For example, the following Java code downcasts to access extension properties:

```
import gw.pl.contact.entity.Address;
import extensions.pc.contact.entity.AddressExt;

...
public void sendAddressProperties ( Address a ) {

    // downcast from Address to AddressExt. See the import statements above.
    AddressExt addressWithExtension = (AddressExt) a;

    // use extension properties from Java...
    System.out.println(addressWithExtension.getMyExtensionProperty());
}
```

After modifying the data model configuration in ways that affect extension properties, remember to regenerate the Java libraries. See “Regenerating Java API Libraries” on page 633.

The Java entity interfaces are supported only in Java code. From Gosu, use the syntax `entity.ENTITYNAME`, or where unambiguous just type the entity name.

Entity Interfaces for Completely Custom Entity Types

If you create custom entity types (rather than extend existing ones), PolicyCenter creates a customer entity interface with the following Java fully-qualified name:

`PACKAGE_PREFIX.SUBPACKAGE.entity.ENTITYNAME`

By default:

- `PACKAGE_PREFIX` is `extensions.pc`, though can be configured. See “Configuring Entity and Typelist Fully-Qualified Names from Java” on page 637.
- `SUBPACKAGE` is defined by the `subpackage` attribute in the entity declaration (or the default) in the data model configuration file. If it is omitted, a default package is used. See “Configuring Entity and Typelist Fully-Qualified Names from Java” on page 637.

The Java entity interfaces are supported only in Java code. From Gosu, use the syntax `entity.ENTITYNAME`, or where unambiguous just type the entity name.

Configuring Entity and Typelist Fully-Qualified Names from Java

You can control the Java package names for entity types that you add in PolicyCenter data model configuration XML files. As mentioned earlier, the generated entity package includes a package prefix and a subpackage using the following pattern for fully-qualified entity type names:

`PACKAGE_PREFIX.SUBPACKAGE.entity.ENTITYNAME`

There are several ways to configure the package of entity types in Java, as described in the following table.

Package configuration for Java entity or type-list types	How to configure	Affects types in the base configuration and not in the extensions folder	Affects types that contain customer extensions	Affects types that you add, including typelists and subtypes of existing entities
custom subpackage	<p>In an entity type or typelist declaration, there is an attribute called subpackage. Set this attribute to a custom subpackage.</p> <p>If you omit the subpackage attribute, PolicyCenter uses a default subpackage. See later in this table regarding configuring the default subpackage.</p>	No	No	Yes
default subpackage	<p>If you omit the subpackage attribute on the entity or typelist definition, PolicyCenter uses a default subpackage. To change the default subpackage, edit the file <code>extensions.properties</code> in Studio. Change the property <code>gw.pl.metadata.codegen.package.default.subpackage</code> to your desired subpackage.</p> <p>In the base configuration, the default subpackage is <code>policy</code>.</p> <p>In the base configuration of <code>ContactManager</code>, the default subpackage is <code>contact</code>.</p>	No	No	Yes
package prefix	<p>You can change the package prefix from Java for entity types that you create by editing the file <code>extensions.properties</code> in Studio. Change the property <code>gw.pl.metadata.codegen.package.prefix</code> to the desired prefix. In the default configuration, the package prefix is <code>extensions.pc</code>.</p> <p>This package prefix is used for generating entity type interfaces for two cases:</p> <ul style="list-style-type: none"> • Entity types that you originally create • Customer extension entity interface, which PolicyCenter generates when you extend an existing entity type with new properties. See “Customer Extension Entity Interface” on page 636. <p>In contrast, PolicyCenter ignores this package prefix configuration when generating:</p> <ul style="list-style-type: none"> • The base entity interface • The core entity interface <p>For typelists, the same rules apply for the package prefix in typelist classes. PolicyCenter uses the package prefix in generating typelist types for:</p> <ul style="list-style-type: none"> • Typelists that you originally create • Customer extension typelist classes, which PolicyCenter generates when you extend an existing typelist. See “Typecode Classes from Java” on page 638. <p>Warning: In the file <code>extensions.properties</code>, there is an additional package prefix <code>gw.pl.metadata.codegen.package.prefix.internal</code>. That is for internal use only. Do not change it.</p>	No	Yes	Yes

Typecode Classes from Java

The Java API exposes typelists and typecodes as Java classes, in contrast to entity types which PolicyCenter exposes as interfaces.

To access a typecode, first get a reference to the appropriate typelist class. The package naming is similar to the pattern for entity data, with some differences. See “Entity Packages and Customer Extensions from Java” on page 634.

PolicyCenter exposes each typelist type as several classes, which the following table describes:

- For the rightmost column, *TL* represents the typelist name, and *SUBPACKAGE* represents the subpackage. The typelist type declaration `subpackage` attribute specifies the *SUBPACKAGE*. This attribute only exists on the original `.tti` file that declares the typelist.
- If the typelist type declaration omits the `subpackage` attribute, a default is used based on whether the typelist is a *platform typelist*. If the typelist declaration attribute `platform` has value `true`, the typelist is a Guidewire platform typelist.
- Whether a typelist is a platform typelist also affects the package prefix. Refer to the table for details.

Terminology	Description	Fully-qualified type name
Typelists that Guidewire defines		
<i>base typelist class</i>	This class contains only the base configuration typecodes. All typelists that Guidewire creates have a base typelist class. This class contains typecodes in the following data model files: <code>.tti</code>	<p>For platform typelists:</p> <ul style="list-style-type: none"> The class is <code>gw.pc.<i>SUBPACKAGE</i>.typekey.<i>TL</i></code> The default subpackage is <code>platform</code> and is not configurable. <p>For non-platform typelists:</p> <ul style="list-style-type: none"> The class is <code>gw.pc.<i>SUBPACKAGE</i>.typekey.<i>TL</i></code> The default subpackage is <code>policy</code>. For ContactManager, the default subpackage is <code>contact</code> The default subpackage is configurable in the <code>extensions.properties</code> file. See “Configuring Entity and Typelist Fully-Qualified Names from Java” on page 637 <p>For example, for platform typelist <code>BatchProcessType</code>:</p> <ul style="list-style-type: none"> Class name is <code>BatchProcessType</code> Package is <code>gw.pc.batchprocessing.typekey</code>
<i>internal extension typelist class</i>	This class contains base typecodes and application-specific typecodes. This class exists only if the Guidewire platform defines a base version but PolicyCenter provides additional typecodes for application-specific use. This class contains typecodes in the following data model files: <code>.tti</code> , <code>.tix</code>	<p>For all internal extension typelist classes:</p> <ul style="list-style-type: none"> The class is <code>gw.pc.<i>SUBPACKAGE</i>.typekey.<i>TLConstants</i></code> The subpackage is defined by the subpackage of the typelist this extends. If it is undeclared, the default is <code>platform</code> and is not configurable. <p>For example, for platform typelist <code>BatchProcessType</code>:</p> <ul style="list-style-type: none"> Class name is <code>BatchProcessTypeConstants</code> Package is <code>gw.pc.batchprocessing.typekey</code>

Terminology	Description	Fully-qualified type name
Typelist extensions and new typelists		
<i>customer extension typelist class</i>	This class contains base typecodes, application-specific typecodes, and customer typecodes. This class exists only if there are customer data model extensions. If an internal extension typelist class exists for that typelist type, the customer extension typelist class includes everything from the internal extension typelist class. Otherwise, the customer extension typelist class includes typecodes from the base typelist class. This class contains typecodes in the following data model files: .tti, .tix, .tx.	<p>For all customer extension typelist classes:</p> <ul style="list-style-type: none"> The class is: <code>PACKAGE_PREFIX.SUBPACKAGE.typekey.TLExtConstants</code> <code>SUBPACKAGE</code> is defined by the subpackage of the typelist this extends. Refer to the row in this table for the <i>base typelist class</i>. By default, <code>PACKAGE_PREFIX</code> is <code>extensions.pc</code> but is configurable. See “Configuring Entity and Typelist Fully-Qualified Names from Java” on page 637 <p>For example, for platform typelist <code>BatchProcessType</code>:</p> <ul style="list-style-type: none"> Class name is <code>BatchProcessTypeExtConstants</code> Package is <code>extensions.pc.batchprocessing.typekey</code>
<i>customer typelist class</i>	This class contains all typecodes. All typelists that you create have this class. Contains typecodes in the following data model files: .tti.	<p>For all customer typelist classes:</p> <ul style="list-style-type: none"> The class is <code>PACKAGE_PREFIX.SUBPACKAGE.typekey.TL</code> The default subpackage is <code>policy</code>. For <code>ContactManager</code>, the default subpackage is <code>contact</code> The default subpackage is configurable in the <code>extensions.properties</code> file. See “Configuring Entity and Typelist Fully-Qualified Names from Java” on page 637 By default, <code>PACKAGE_PREFIX</code> is <code>extensions.pc</code> but is configurable. See “Configuring Entity and Typelist Fully-Qualified Names from Java” on page 637. <p>For example, for typelist <code>Example</code> with omitted subpackage:</p> <ul style="list-style-type: none"> Class name is <code>Example</code> Package is <code>extensions.pc.policy.typekey</code>

Getting Typecodes Using the get method

The static properties on a typelist that represent a typecode have the `TC_` prefix, just like from Gosu. However, to actually work with the typecode, you must call the `get` method on the static property. The `get` method returns the appropriate typecode object.

For example, suppose you want the typecode `Approval` from a fictional example typelist called `ExampleType`. Assume that the typelist is declared with the example subpackage `testsub`.

- If the type is in the base configuration for multiple Guidewire applications, use the syntax:

```
tc = gw.pl.testsub.typekey.ExampleType.TC_APPROVAL.get()
```

- If the type is defined in the base configuration and extended for PolicyCenter, use the syntax:

```
tc = gw.pc.testsub.typekey.ExampleTypeConstants.TC_APPROVAL.get()
```

- If the type is defined in a customer data model extension, use the syntax:

```
tc = extensions.pc.testsub.typekey.ExampleTypeExtConstants.TC_APPROVAL.get()
```

If you forget to call the `get` method, the resulting object is inappropriate for comparing typecodes, as well as most other contexts. For related information, see “Comparing Entity Instances and Typecodes” on page 641.

Wherever possible, use the static constants for typecode values. Using the typecode static constants ensures that your code is type safe. Mistakes in the typecode string are caught as compile errors rather than only at run time. If the code is known at compile time, always use the static constants to get the typekey object.

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. Be warned that the `getTypeKey` method uses reflection, and typecode values cannot be validated at compile time. For example:

```
tc = gw.pl.testsub.typekey.ExampleType.getTypeKey(typeCodeString)
```

Entity Subtype Typelists

In addition to standard typelists, there are special typelists that exist only to distinguish *subtypes* of entities. These *subtype typelists* help PolicyCenter specify the entity subtype in each database row.

The typelist name matches the name of the entity type. Based on this typelist name, PolicyCenter creates the typelist classes for the Java API using the same basic pattern as regular typelists in “Typecode Classes from Java” on page 638.

If the entity subtype is shared among multiple Guidewire applications, the Java class name is:

```
gw.pl.SUBPACKAGE.typekey.ENTITYNAME
```

If PolicyCenter overrides the entity subtype, there is a typelist class with the Java class name:

```
gw.pc.SUBPACKAGE.typekey.ENTITYNAMEConstants
```

If you extend an entity subtype of a pre-existing entity, there is a typelist class with the Java name:

```
PACKAGE_PREFIX.SUBPACKAGE.typekey.ENTITYNAMEExtConstants
```

If you created the entity subtype, there is a typelist class with the Java name:

```
PACKAGE_PREFIX.SUBPACKAGE.typekey.ENTITYNAMEExtConstants
```

Comparing Entity Instances and Typecodes

To compare entity instances for equality, always use the `entity.equals(otherEntity)` method. Using the `==` (double equals) operator is unsafe to test for equality.

To compare typecode values for equality, similarly use the `equals` method. However, be sure that you have a reference to the actual typecode class using the `get` method as mentioned in “Typecode Classes from Java” on page 638.

For example, suppose that the code `claim.getState()` returns a typecode type called `PolicyState` and you want to compare the result to a specific value.

It is incorrect to do either of the following because the Java `==` (double equals) operator is unsupported for comparing typecodes:

```
policy1.getState() == policy2.getState()  
policy.getState() == PolicyState.TC_OPEN.get()
```

The following line is incorrect because it omits the `get` method after getting a static instance by name:

```
policy.getState().equals(PolicyState.TC_OPEN)
```

Instead, use the following expression syntax:

```
policy.getState().equals(PolicyState.TC_OPEN.get())  
policy1.getState().equals(policy2.getState())
```

Be very careful with code that compares two entities or two typecodes.

Entity Bundles and Transactions from Java

Before writing Java code, regenerate the Java API libraries. This process ensures that any data model changes or extensions are in the most recently generated libraries. See “Regenerating Java API Libraries” on page 633.

If you implement a Java plugin, see “Plugin Overview” on page 123 and “Special Notes For Java Plugins” on page 130.

Getting a Reference to an Existing Bundle in Java

To use entity instances, in many cases you need a reference a *bundle*. A bundle is a programmatic abstraction that represents one database transaction. See “Bundles and Database Transactions” on page 337 in the *Gosu Reference Guide*.

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

WARNING There are many contexts in which it is unsafe to commit a bundle. For example, all plugin code and most PCF code. For more information, see “Bundles and Database Transactions” on page 337 in the *Gosu Reference Guide*.

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.

New Bundles In 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. If you have questions, please contact Guidewire Customer Support.

WARNING There are many contexts in which it is unsafe to commit a bundle. For example, all plugin code and most PCF code. For more information, see “Bundles and Database Transactions” on page 337 in the *Gosu Reference Guide*.

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` API, 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, with an anonymous class instead of a Gosu block. For example:

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

Also see “Running Code in an Entirely New Bundle” on page 347 in the *Gosu Reference Guide*.

Creating New Entity Instances from Java

The recommended API for creating an entity instance is to call the `newInstance` method on the entity type’s `TYPE` property. Pass a bundle reference as a method argument. Depending on whether you need customer extension properties, the syntax varies. For discussion of the two separate interfaces for each entity type, see “Entity Packages and Customer Extensions from Java” on page 634.

From Java plugin code, there is a current bundle. You can use the `Transaction.getCurrent()` method to get the current bundle if one exists.

If there is a current bundle, create a new `Address` entity instance using the following code:

```
import gw.pl.persistence.core.Bundle;
import gw.pl.contact.entity.Address;
import extensions.pc.contact.entity.AddressExt;
import gw.transaction.Transaction;

...
Bundle b = Transaction.getCurrent();

// if you need only the base entity interface properties and methods
Address a1 = Address.TYPE.newInstance(b);

// if you need customer extension properties, downcast to the customer extension interface...
AddressExt a2 = (AddressExt) Address.TYPE.newInstance(b);
```

Alternative API for New Instances

Although not recommended for typical use, you can also create a new entity instance with the `newBeanInstance` method on the `Bundle` class as shown in the following example:

```
Bundle b = Transaction.getCurrent();
Address a1 = (Address) b.newBeanInstance(Address.TYPE.get());
```

This approach requires explicit downcasting from the base class of all entity types and thus it is easier to make mistakes that cannot be caught at compile time.

Getting and Setting Entity Properties from Java

Entity properties appear from Java as getter and setter methods. Getter and setter methods are methods to get or set properties with names that start with `get` or `set`. For example, a readable and writable property named `MyField` appears as the methods `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` or `boolean`, it appears in the interface as `isMyField` instead of `getMyField`.

Examples:

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

If the property is an extension property, you must use the entity extension interface as discussed in “Entity Packages and Customer Extensions from Java” on page 634. Read that topic carefully for the fully-qualified names of the interfaces and a code example of downcasting.

Calling Entity Object Methods from Java

Most entity methods on entity instances appear as regular methods, for example:

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

In some cases you need to know how the method is declared before you can write the necessary Java code:

- The base entity interface may declare the method, in which case no special downcast is necessary.
- The core extension entity interface may declare the method, in which case downcast to that interface or the customer extension interface. See “Accessing Entity Properties and Methods With Base and Core Extension Interfaces” on page 635.
- The customer extension entity interface may declare the method, in which case you may need to downcast to the customer extension interface. See “Customer Extension Entity Interface” on page 636.
- A Gosu enhancement may declare the method. See “Gosu Enhancement Properties and Methods in Java” on page 645.

Querying for Entity Data in Java

In the Java API, if you need to find entity instances, use the query builder API. See “Overview of the Query Builder APIs” on page 129 in the *Gosu Reference Guide*.

The Java API syntax to those classes is nearly identical to the syntax in Gosu for APIs that do not use Gosu blocks.

For some of the query API that uses Gosu blocks, there is no direct Java equivalent.

For other APIs that use blocks as arguments or return values, there are special Java-specific variants of the methods. Contact Guidewire Customer Support for details.

Accessing Gosu Classes from Java Using Reflection

From Java, you must 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 of accessing 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 or Gosu, use the utility class `gw.lang.reflect.ReflectUtil`. The `ReflectUtil` class contains various APIs to get information about a class, get information about features (properties and methods), and invoke object methods or instance methods.

For example, the following Java code calls a static method on a Gosu class using the `ReflectUtil` method `invokeStaticMethod`.

The Gosu class definition:

```
package test1

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

To call the static method from Java with the argument "hello", use the following code:

```
package test1;

import gw.lang.reflect.ReflectUtil;
import gw.transaction.Bundle;

public class MyClass {
    public void doIt() {

        // call a static method on a Gosu class
        String r = (String) ReflectUtil.invokeStaticMethod("test1.MyGosuClass", "myMethod", "hello") ;

        // print the return result
        System.out.print(r);
    }
}
```

If you write your own Gosu class and you want to call methods on it from Java, there is pattern that increases the degree of type safety. See the following procedure.

To make your Java code that calls Gosu classes more typesafe

1. Create a Java interface containing only the set of 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`.
2. Create a Gosu class that implements that interface. It can contain additional methods if desired if they are not needed from Java.

3. In code that needs to get a reference to the object, use `ReflectUtil` to create an instance of the desired Gosu class. There are several approaches:

- Create an instance using `ReflectUtil`.
- Alternatively, define a method on an object that creates an instance. Next, call that method with `ReflectUtil`.

In both cases, at compile time any new object instances returned by `ReflectUtil` have the type `Object`. At run time, downcast to your new Java interface and assign to a new variable of that interface type.

4. You now have an instance of an object that conforms to your interface. Call methods on it as you normally would from Gosu. The getters, setters, and other method names are defined in your interface, so the method calls are typesafe. For example, the Java compiler can protect against misspelled method names.

Gosu Enhancement Properties and Methods in 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. See “Enhancements” on page 231 in the *Gosu Reference Guide*.

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.

For more information about reflection and the `gw.lang.reflect.ReflectUtil` class, see “Accessing Gosu Classes from Java Using Reflection” on page 644.

Class Loading and Delegation for non-OSGi Java

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 follows the rules in the following list to load Java classes, choosing the first rule that matches and then skipping the rules listed after it:

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`, then PolicyCenter delegate loads in general, but there are some internal classes that locally load.

WARNING Java code 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 package `com.guidewire`.

3. All your classes

Any remaining non-internal classes load locally.

WARNING Java classes that you deploy must **never** 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, then the ClaimCenter class loader attempts to load the class locally.

Deploying Non-OSGi Java Classes and JARs

The following deployment instructions work for non-OSGi Java class files or JAR files, independent 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. If you are deploying an OSGi plugin, see the separate section “OSGi Plugin Deployment with IntelliJ IDEA with OSGi Editor” on page 647.

Note: This section discusses the deployment options for the Java API introduced in version 8.0. If you are using the deprecated Java API from PolicyCenter 7, see “Important Changes for Java Code” on page 80 in the *New and Changed Guide*.

Place your non-OSGi Java classes and libraries in the following locations. Any subdirectories must match the package hierarchy. The `shared` directory works for any non-OSGI Java code.

If the class implements a PolicyCenter plugin interface, you can define a separate plugin directory in the PolicyCenter Studio in the Plugins registry for that interface. In the following paths, `PLUGIN_DIR` represents 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`. For more about defining plugin directories in PolicyCenter Studio, see “Adding an Implementation to a Plugins Registry Item” on page 132 in the *Configuration Guide*. Also see “Registering a Plugin Implementation Class” on page 126.

If any Gosu class calls your Java code, for the `PLUGIN_DIR` value you can also use the value `Gosu` instead of `shared`. Be careful to notice the capitalization of `Gosu`.

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

```
PolicyCenter/modules/configuration/plugins/shared/basic/classes  
PolicyCenter/modules/configuration/plugins/PLUGIN_DIR/basic/classes // for Java plugin code only
```

For example, for a class file with fully qualified name `mycompany.MyClass`, create files at one of the following:

```
PolicyCenter/modules/configuration/plugins/shared/basic/classes/mycompany/MyClass.class  
PolicyCenter/modules/configuration/plugins/PLUGIN_DIR/basic/classes/mycompany/MyClass.class
```

Place your libraries (JAR files) and any third-party libraries in the following locations:

```
PolicyCenter/modules/configuration/plugins/shared/basic/lib  
PolicyCenter/modules/configuration/plugins/PLUGIN_DIR/basic/lib // for Java plugin code only
```

OSGi Plugin Deployment with IntelliJ IDEA with OSGi Editor

For a summary of the IDE options for OSGi plugin development, see “IDE Options for Plugin Development in the Java Language” on page 631.

This topic describes specific steps for OSGi plugin development:

- “Generate Java API Libraries” on page 647
- “Launch IntelliJ IDEA with OSGi Editor” on page 647
- “Create an OSGi-compliant Class that Implements a Plugin Interface” on page 648
- “Compiling and Installing Your OSGi Plugin as an OSGi Bundle” on page 649
- “Using Third-Party Libraries in Your OSGi Plugin” on page 651

Generate Java API Libraries

Before starting work with Java and OSGi, regenerate the PolicyCenter Java API libraries with the `regen-java-api` tool. See “Regenerating Java API Libraries” on page 633. This is a requirement that is independent of which Java IDE that you use.

Launch 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 `PolicyCenter/bin` directory and type the following:

```
gwpc plugin-studio
```

IMPORTANT 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.

Creating a New Project With OSGi Plugin Module

1. If this command is executed for the first time for one Guidewire product, IntelliJ starts with an empty workspace and no current project
2. To create a new project, select **Create New Project**. In the module type list, click **OSGi Plugin Module**. In the **Project name** field, type the project name. In the **Project location** field, type the project location. Do not yet click **Finish**.
Alternatively, if you want to add or import an OSGi module to an existing empty project, in the **Project Structure** dialog, select **Empty Project** and set the **Project name** field. Next, add a module in the **Project Structure** dialog by clicking the plus sign (+). Choose either **New Module** (for new module) or **Import Module** (to select another module to import). For a new module, select the module type **OSGi Plugin Module**. In the **Module name** field, type the module name. Continue to follow the rest of this procedure.
3. Open the **More Settings** pane, which may be initially closed. Set the name of the new module as appropriate. By default, the **Bundle Symbolic Name** field matches the name of the module. You can optionally change the symbolic name to a different value in this dialog. The bundle symbolic name defines the main part of the output JAR file name before the version number. You can also optionally change the version of the bundle in the **Bundle Version** field.
4. Click **Finish**.

5. 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 JDK picker to your Java 7 JDK, which might be labelled 1.7.
 - b. If there is no Java 7 JDK listed, click the New... button. Click JDK, then select your Java 7 JDK on your disk. Next, set the Project JDK picker to your newly-created Java 7 JDK configuration.
6. For advanced OSGi settings, see “Advanced OSGi Dependency and Settings Configuration” on page 654.

Create an OSGi-compliant Class that Implements a Plugin Interface

Follow the procedure below to implement an OSGi plugin with IntelliJ IDEA with OSGi Editor.

To implement a new OSGi plugin

1. Remember to regenerate the Java libraries. See “Generate Java API Libraries” on page 647.
2. In IntelliJ IDEA with OSGi Editor, navigate under your OSGi module to the `src` directory.
3. Create new subpackages as necessary. Right-click on `src` and choose **New → Package**. To follow along with an example of a simple startable plugin, create a `mycompany` package to contain your new classes.
4. Right-click on the desired package for your class.
5. Choose **New → New OSGi plugin**. IntelliJ IDEA with OSGi Editor opens a dialog.
6. In the **Plugin class name** field, enter the name of the Java class that you want to create. For our example, type `DemoStartablePlugin`.
7. In the **Plugin interface** field, enter a fully qualified name of the plugin interface you want to implement. Alternatively, to choose from a list, click the ellipsis (...) button. Type some of the name or scroll to the desired plugin interface. Select the desired interface and then click **OK**.
8. IntelliJ IDEA with OSGi Editor displays a dialog **Select Methods to Implement**. By default, all methods are selected. Click **OK**.
9. 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. See “Creating a New Project With OSGi Plugin Module” on page 647. 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.
For example, one OSGi bundle can encapsulate messaging code for a messaging destination. A messaging destination may implement the `MessageTransport` interface. The same messaging destination may optionally also implement the `MessageReply` and `MessageRequest` plugin interfaces. If the multiple plugin implementations have shared code and third-party libraries, you could deploy them in the same OSGi bundle.

Example Startable Plugin in Java Using OSGi

The following example defines a class that implements the `IStartablePlugin` interface. The following class simply prints a message to the console for each application call to each plugin method. Use the following example to confirm 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;
    }
}
```

Compiling and Installing Your OSGi Plugin as an OSGi Bundle

The main script for OSGi compilation and deployment is an Ant build script. See “Advanced OSGi Dependency and Settings Configuration” on page 654.

The Ant build script does the following:

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. For related information, see “Advanced OSGi Dependency and Settings Configuration” on page 654. The script copies your final JAR file to the following PolicyCenter directory:
`PolicyCenter/modules/configuration/deploy/bundles`

To compile and install your OSGi plugin implementation as an OSGi bundle implementation

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 it succeeds, it prints:

BUILD SUCCESSFUL

3. Switch to or open the regular PolicyCenter Studio application, 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.

IMPORTANT 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. See “Updating Your OSGi Plugin Project After Product Location Changes” on page 655.

4. In PolicyCenter Studio (not IntelliJ IDEA with OSGi Editor), 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 on the item **registry**, and choose **New → Plugin**.
 - b. In the plugin dialog, enter the plugin name in the **Name** field. For our example, use **DemoStartablePlugin**.
For plugin interfaces that only support one implementation, just enter the interface name without the package. For example, **IStartablePlugin**. The text 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 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 our 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 it, enter the interface name without the package.
For our 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 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 our 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.

IMPORTANT For more information about the Plugins registry, see “Plugin Overview” on page 123

- h. Make whatever other changes you need to make in the regular PolicyCenter Studio application. For example, if your plugin is a messaging plugin, configure a new messaging destination. See “Using the Messaging Editor” on page 153 in the *Configuration Guide* and “Messaging and Events” on page 289. You may need to make other changes such as changes to your rule sets or other related Gosu code.
- i. Start the server.

To run the server from Studio, if PolicyCenter Studio was 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 `PolicyCenter/bin` directory and type:

```
gwpc dev-deploy
```

Next, start the server from Studio, such as clicking the **Run** or **Debug** menu items or buttons.

Alternatively, run the QuickStart server from the command line. Open a command prompt in the `PolicyCenter/bin` directory and type the following:

```
gwpc dev-start
```

An earlier topic showed an example startable plugin. See “Create an OSGi-compliant Class that Implements a Plugin Interface” on page 648. If you used that example, closely read the console messages during server startup. The example OSGi startable plugin prints messages during server startup. This proves that PolicyCenter successfully called your OSGi plugin implementation.

Using Third-Party Libraries in Your 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 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. See “Embed a Third-Party Java Library in your OSGi bundle” on page 651.
- **Deploy as a separate OSGi bundle (requires 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. See “Deploying a Third-party OSGi-compliant Library as a Separate Bundle” on page 651

Deploying a Third-party OSGi-compliant Library as a Separate Bundle

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.

To use third-party OSGi-compliant libraries separate from your OSGi plugin bundle

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 there are no compile errors.
4. Open a command prompt at the root of your module and type:

```
ant install
```

The tool generates various messages, and concludes with:

```
BUILD SUCCESSFUL
```

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

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 may require special configuration for how to import the Java packages that your embedded libraries reference.

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

It may be that some of the classes in your third-party library but that you do not use have dependencies on external classes. The library might use an API that relies on classes that are unavailable at run time. OSGi tries to avoid this risk by ensuring at server startup that all required classes exist. Even if you never call those APIs, there will be errors on server startup because OSGi tries to verify all libraries are available, including ones you never called.

Fortunately, you can modify the configuration file to ignore unavailable packages during OSGi library validation phase on server startup. The following instructions include techniques to mitigate these problems.

To use third-party libraries in your OSGi plugin

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 compile errors.
4. Open a command prompt at the root of your module and type:
`ant dist`
The tool generates various messages, and concludes with:
`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.

WARNING Never directly change the file `MANIFEST.MF`. It is auto-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 “exclude”. Finally 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`, set the line as follows:

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

8. Deploy your OSGi plugin and test it. Look for server startup errors that look similar to:

```
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"
```

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

For more information about the `bnd.bnd` file, see “Advanced OSGi Dependency and Settings Configuration” on page 654.

Example of Embedding Third-Party Libraries in an OSGi Plugin

Suppose your OSGi plugin requires the Java library called Guava, specifically version 15. First, download `guava-15.0.jar` from the project web site. Next, move the `guava-15.0.jar` file to the `inline-lib` folder within your OSGi project within IntelliJ IDEA with OSGi Editor.

Next, write some Java code that uses this library. For example:

```

import com.google.common.escape.Escaper;
...
// use the class com.google.common.escape.Escaper in guava-15.0.jar
Escaper escaper = XmlEscapers.xmlAttributeEscaper();
s = escaper.escape(s);

```

From a command prompt, run the following command:

```
ant dist
```

That tool generates OSGi metadata and prints various messages. If successful, the final line is:

```
BUILD SUCCESSFUL
```

Open the file MODULE_ROOT/generated/META-INF/MANIFEST.MF and check that Import-Package header does not include any unexpected packages. Our example file might look like the following:

```

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: 1.7.0_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-1.7

```

Note the following line:

```
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 runtime. 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.

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. Finally, add an asterisks(*) 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, set the line as follows:

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

Run the "ant install" tool again. The scripts embed the guava JAR in your OSGi bundle.

Open the MANIFEST.MF file and notice two changes:

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

For example:

```

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: 1.7.0_45 (Oracle Corporation)
Bundle-RequiredExecutionEnvironment: JavaSE-1.7
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

```

With this change, the server now starts up with no OSGi class loading errors.

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.

Dependencies

The IntelliJ IDEA with OSGi Editor auto-creates the following dependencies:

- `PROJECT/MODULE/inline-lib` directory – Directory for third-party Java libraries to embed in your OSGi bundle. See “Embed a Third-Party Java Library in your OSGi bundle” on page 651.
- `PROJECT/MODULE/lib` directory – Directory for third-party OSGi-compliant libraries to use but deploy into a separate OSGi bundle. See “Deploying a Third-party OSGi-compliant Library as a Separate Bundle” on page 651.
- `PolicyCenter/java-api/lib` – The PolicyCenter Java API files that the `regen-java-api` tool creates.

If you have any JAR files that you require, put them in the `lib` or `inline-lib` directories, as specified 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.

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 this 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 configures 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, this file looks like:

```

Import-Package=*
DynamicImport-Package=
Export-Package=
Service-Component=*
Bundle-DocURL=
Bundle-License=
Bundle-Vendor=
Bundle-RequiredExecutionEnvironment=JavaSE-1.7
-consumer-policy=${range;[==,+)}

```

```
-include=build.properties
```

For the format of this file, see:

<http://www.aqute.biz/Bnd/Format>

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

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.

Updating Your OSGi Plugin Project After Product Location Changes

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.

To update your OSGi project after Product Location Changes

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. See “Launch IntelliJ IDEA with OSGi Editor” on page 647.
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. See “Compiling and Installing Your OSGi Plugin as an OSGi Bundle” on page 649.

